SERVER-BASED DATA PUSH ARCHITECTURE FOR DATA ACCESS PERFORMANCE OPTIMIZATION

BY

SURENDRA BYNA

DEPARTMENT OF COMPUTER SCIENCE

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Department of Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved _____
Adviser

Chicago, Illinois
December 2006

ACKNOWLEDGEMENT

It has been a long journey to get to this point and it would not have been possible without the cooperation, advice, help, and patience of numerous people around me. It is my pleasure to take this opportunity to thank all those who have contributed to making my professional and personal life better.

It is difficult to overstate my gratitude to my Ph.D. advisor, Professor Xian-He Sun. He has been the motivation and guiding force for me to take up this path. Without his consistent support and advice this work would not have been possible. I heartily thank Dr. Sun for giving me freedom to research, for guiding me into a right direction, and for his personal care.

I am extremely thankful to Dr. Rajeev Thakur and Dr. Bill Gropp, for encouraging me and giving me excellent ideas for research. I look forward to working with them in the future beyond this dissertation. I thank Dr. Kirk Cameron for his advice, and Dr. Hong Zhang for her encouragement.

I thank my thesis committee members Dr. Zhiling Lan, Dr. Cynthia Hood, Dr. Dietmar Rempfer, and Dr. James Stine for providing valuable feedback and suggestions. Dr. Lan has been helpful in giving guidance at group meetings of SCS Laboratory, for which I am very thankful to her.

I thank all our SCS lab members for their friendship, thoughts and suggestions in many aspects of my life. I thank Ming Wu for his help in numerous occasions. Cong Du has been a great friend through out my stay at the lab. Conversations with Nehal Mehta have always been entertaining and informative. I thank Yong Chen for his collaboration in my research work and for being a very nice friend. I also thank Yawei Li, Xiaoshan

He, Petre Brotea, Kasidit, Luciano Piccoli, Gregor Tamindza, and John Sohn for their criticism in my research and friendship.

Numerous friends helped and encouraged me through out this journey. I am grateful to Sairam and Srinivas Chinnam for their support and keeping in touch with my parents often. I thank Shinta for being very caring. I appreciate Rajasekar Karawalla, Rahul Bhuman, Ravi Yeluguri, Deepthi, Prasanth Veera, Hareesh Achi, Madhavi, Vinay Kudithipudi, Sudheer Palsani, Srinivas Banda, and Hiranmayee for their support in many ways. I also thank all my other friends who helped in developing as a better human being.

Without the help of my relatives, I cannot imagine my being at this stage. Especially, I am thankful to Ravindra bava, Venu, Murali, Venkateswarlu uncle, Sujatha aunty, and Manjula akka for being there at all occasions in our family and for helping my parents from time to time. I appreciate my cousins Purnima, Anupama, Rajesh, Srikanth, Sudhir, Venkateswarlu, and Subhashini for their friendship. I take this opportunity to express my gratitude to Raghaviah uncle for his priceless advice, which I hope I could use in each step of my life, and Uma aunty for her loving support. Finally and most importantly, I cannot thank my family enough for their unconditional love. My brother-in-law, Hima Kumar and his family members have been very accepting and loving. My sister, Jyothi has been cool and cheering on for my success. My parents, Suguna and Parabrahmam Byna have been patient with me. They have been enormously supportive of me through out my life. They made sure I get quality education for my career and for my life. For all they have done to me in my growth, I dedicate this dissertation to my parents. I hope I could do more for them in the future. I thank the forces of nature and God for keeping me healthy and mentally strong.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

Symbol	Definition
HEC	High-End Computing
MPI	Message Passing Interface
SSP	Sustained System Performance
DSM	Distributed Shared Memory
SMP	Shared Memory Parallelism
SMAC	Simple Memory Access Cost
CBE	Cell Broadband Engine
DAH	Data Access History
DPS	Data Push Server
DPS-P	Data Push Server for Parallel Computing
PFE	Prefetching Engine
DAP	Data Access Pattern
DPR	Data Propeller
PDM	Pattern Detection Manager
PSS	Prefetch Strategy Selector
FAS	File Access Pattern
DPM	Dynamic Power Management
DVS	Dynamic Voltage Scaling
NIC	Network Interface Card
MME	Memory Management Engine
MLDT	Multi-level Difference Table
FAS	File Access Server
SOA	Service Oriented Architecture
SPMD	Single Program, Multiple Data

ABSTRACT

In the past several years, High-End Computing (HEC) has seen enormous growth in peek performance, and development of Peta-flop supercomputer is in the near horizon. Despite these advances, data access delay has been a major reason for poor sustained system performance (SSP) on HEC machines. Multiple levels of memory hierarchy have been incorporated into computer architecture to take advantage of locality among data accesses to reduce the gap between peak and sustained performances. However, many applications lack locality, which make these advances inefficient. Researchers have proposed many optimization methods to improve locality and to prefetch data into these cache memories before CPU demands for it. However, there are limitations in applying these methods. First, locality is application dependent and choosing an efficient combination among all existing tuning methods at runtime remains elusive. Second, the current client-initiated prefetching strategies do not work well for applications with complex, non-contiguous data access patterns.

To bridge the performance gap, we introduce server-based data push architecture. In this architecture, a dedicated server named Data Push Server (DPS) initiates and proactively pushes data closer to the processing units in time. We addressed the issues of monitoring data access history, making spatial and temporal access pattern predictions, architecture modifications to push the predicted data values close to processing cores, and modeling data access cost. We have quantified data access cost from communication and middleware latencies. We present analytical models for memory performance prediction based on data access patterns that are useful to choose effective optimization and prefetching strategies with low overhead. We have applied these models to improve the

performance of Message Passing Interface (MPI) derived datatypes. We have studied the server-push architecture by enhancing SimpleScalar simulator with a dedicated processing unit that pushes data for another processor. The simulation results show significant performance gains. Our DPS architecture is extendable to various levels of memory hierarchy, and has a broader impact on high-end computing to improve productivity.

CHAPTER 1

INTRODUCTION

1.1 DATA ACCESS PERFORMANCE

High end computing (HEC) is a major strategic tool for science, engineering, and industry. HEC simulations in various areas of science enable to understand the world around us [Abra03, Kusn05]. They study the universe, enabling us to observe the systems that are too small (nanotechnology, biotechnology, DNA analysis etc.), too large (astrophysics, hurricanes, tsunamis, aircraft, atmosphere etc.), and too dangerous (nuclear weapons) for direct experimental observation. HEC machines have emerged with TeraFlops of computing power, and PetaFlop computing is in the near horizon. The current fastest supercomputer IBM BlueGene/L beta-System has 32768 processors, with a peak performance of 367 TeraFlops [Ibmb04]. Even with the existence of these powerful supercomputers, the demand for more powerful supercomputers continues, and many projects are in development to quench computing power thirst.

However, there is a rapidly growing gap between the peak performance of HEC machines and sustained system performance of applications running on these systems [Dhpcs]. While the peak performance of current HEC machines is improving rapidly, the sustained performance of applications on these machines is in the range of meager 10% (see Figure 1.1). Disparity among performance growth of processors, storage, memory, network, and applications has been the cause of this gap. The performance of processor and network interconnect are improving multiple times faster than that of memory and storage. Among these disparities, poor data access performance is a major reason for the divergence gap. Traditionally the performance has been linked to processor speed. The

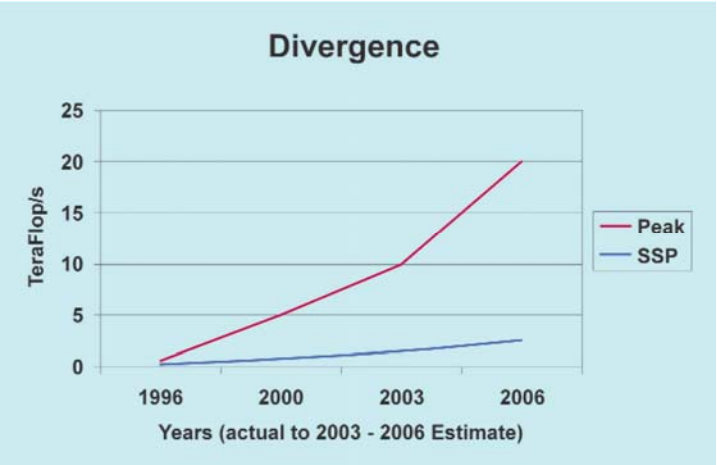


Figure 1.1. Growing gap between peak and sustained performance
Source: HECRTF [Feda04]

capacity and speed of processors doubled every 18 months complying with Moore's Law until 2004, due to the increasing density of transistors within a chip. In contrast, main memory (DRAM) speeds and bandwidth haven't increased enough to catch up with the processor performance. Since 2004, multi-core processor technology is making strides of improvement and the future belongs to these powerful processors. These advances are fueling the performance gap between processing and data access further into new levels. This trend is predicted to continue for the next decade and beyond. The increasing incompatibility between performance of processor and memory has been an obstacle to fully exploit the technological advances and the expected performance from the hardware.

Immense research effort has been spent on reducing the performance gap between processor and memory. Caching is a commonly used method to mask the performance disparity between processing and data access performance. Advanced hierarchical memories that include cache memories at various levels are available to bridge this gap. A cache memory works on the principle of spatial and temporal locality [Smit82], which

stores recently accessed cache lines of data. However, there are many applications that lack locality in accessing the memory. These applications spend a major fraction of execution time waiting for data accesses. In other words, cache memories are exploited better if the cached blocks of data are reused extensively before other cache blocks replace them.

Transforming and reordering the memory accesses improve application performance [Mcki96, Kand99, Vudb01]. As loops are the basic blocks, where most of time is spent in HEC applications, various loop optimization techniques have been developed to enhance the memory hierarchy utilization. Loop transformations (loop unrolling, loop fusion, loop interchange, loop reversal and loop tiling) are some of the most effective loop optimizations. Nevertheless, developers have to be aware of these optimization techniques and the location of applying them to improve the performance of applications.

Prefetching is another strategy to mask the data access latency. While caching holds data temporarily, prefetching brings the data to a cache closer to the computing processor before it is requested. Various prefetching strategies have been proposed and developed during the past decade [Chba95, Ctws01, Fupa91, Jogr97, Kasi02]. Based on the data access history, these strategies try to predict future references using the distances (strides) between sequences of accesses. To predict these strides, algorithms are developed ranging from basic constant stride prediction to complex Markov chain predictions. Prefetching instructions can be issued either by a developer or a compiler at software level or by CPU at hardware level. These instructions have to be issued carefully to avoid negative effects of evicting useful cache lines. Accurate prediction and timely issuing of prefetch instructions increase the effectiveness of prefetching.

1.2 PROBLEM STATEMENT

Data access performance can be improved by utilizing cache memory effectively. Accurate and aggressive predictions that help prefetching data in time and automatic reordering of loops can help reduce the CPU stall time in waiting for data access. This is the path to reduce the divergence gap problem of current HEC machines and to improve their productivity.

Although numerous researchers have proposed hardware and software optimization mechanisms to reduce the processor-memory performance gap, memory access is application dependent. Some advanced compilers utilize these optimization techniques at various levels to improve application performance. However, compilers alone are not sufficient to achieve the best possible optimization due to the dynamic behavior of the memory accesses [Bgst03]. Optimizations that are implemented by hand with the knowledge of optimizations, achieve better performance than compiler optimizations. But superior manual optimizations require extensive knowledge of the hardware architecture and also about the data access patterns of application. The developer needs to be aware of efficient optimization techniques to be applied in the right place. Choosing an effective combination of optimizations at runtime among all existing tuning methods is remaining elusive.

Data prefetching at hardware level is a challenging task. Although many strategies exist in literature, their performance varies largely from application to application, and is generally poor on HEC computers. The poor performance of current prefetching technology may be due to different reasons. One noticeable reason is that current

prefetching is based on client-initiated prefetching, where the computing processor initiates the prefetching. While letting the computing processor to prefetch the required data seems to be a straightforward solution, client-initiated prefetching has many limitations. For instance, predicting what data to fetch require computing power; aggressive (accurate) prediction algorithms may take computing power away from application and therefore reduce the system performance; the prediction information obtained by the computing processor may get lost in the memory hierarchy; the client does not know where the data is and have to compete with other computing processors for data access, therefore cannot perform in-time prefetching. In addition, in many HEC machines, computing processors have reduced OS implementations. They do not have the means to collect data access information effectively for data access prediction and prefetching. Chip-level multiprocessing puts multiple cores share the same data bus and high-level caches, and puts even more pressure on data access if multicore concurrent processing is conducted.

In recognizing the limitation of client-initiated prefetching, and taking the advantage of the abundant computing power, several new prefetching strategies have been proposed recently on multi-core processors [Alke96, Ctws01, Hafe04, Hugh00, Lwwh02, Lukc01, Roso01, Shpc01]. The main idea of these approaches is to let a helper thread run ahead of the program main thread on a separate core to initiate a load cache misses in a multicore machine. The pre-execution can be conducted on a core close to the program's main thread, which is called *pull-based* pre-execution, or on a core close to the memory, which is called *push-based* prefetching [Hafe04, Solt02]. The former fits the current multicore chip architecture well. The latter requires special hardware support but is more efficient

in masking data access delay and also realizes the decoupling of data access as suggested by Smith [Smit82].

1.3 OUR APPROACH

Caching and prefetching techniques enhance data access performance, but they must be applied in HEC application development efficiently. As the time-to-solution of an application includes the time to find a set of optimization parameters, we have to reduce the time to search for optimization parameters at runtime. Prediction of what future data will be used by an application and timing of its usage is important to make prefetching effective. Prefetching incorrect data, either too early or too late has adverse effects on performance. In our research, we introduce strategies to improve caching and prefetching and overall, to bridge the gap between peak performance and sustained system performance of HEC.

We extend the concept of decoupling of computing and data access. We design a data access server system, named Data Push Server (DPS), dedicated to predict data access pattern and to push data closer to computing processors in time. Here the term ‘push’ also means that, unlike traditional client-initiated prefetching, DPS initiates prefetching. DPS does not conduct any computing or pre-execution. Its whole purpose is to providing data push service and to prefetch based on data access prediction. This further separation of computing and data service has several benefits. First, a dedicated server can adapt to complex prediction algorithms for more aggressive prediction and can push data into multiple computation threads or cores. This is especially beneficial for HEC, where parallel processing is often achieved with the SPMD model [Cull97, Cusg98]. Second,

DPS is flexible to choose strategies dynamically to predict future accesses based on data access history. Instead of looking for a single magical prediction method for all data access patterns, which does not exist, DPS can adaptively choose a prediction method based on the history of accesses and compiler hints. This, again, is very beneficial to HEC where few of the so called “grand challenge applications” often running repeatedly. Third, we use temporal data access information to predict *when* to push data. This avoids costly synchronization needed for pre-execution strategies to initiate prefetching *in time*. DPS can be implemented at various levels of a memory hierarchy.

We have also developed models to classify the data access delay in message passing and to predict that cost based on data access patterns. In message passing, data access delay has become a major portion as the network speeds have improved quite faster compared to memory access performance. To improve the data access cost, we first classified it based on non-contiguity and the size of messages. We then developed a model to predict memory access cost, in order to choose optimization parameters that improve cache utilization. In this process, we used various cache optimization techniques including array padding, cache blocking, software prefetching, and loop unrolling. We applied these models in improving the performance of derived datatypes in the Message Passing Interface implementation MPICH2. MPI derived datatypes allow users to describe noncontiguous memory layout and communicate noncontiguous data with a single communication function. This feature enables an MPI implementation to optimize the transfer of noncontiguous data. Our automatic optimized implementation achieves performance closer to which an advanced developer can achieve by packing/unpacking

noncontiguous messages with optimizations. This is a significant performance improvement over original implementation of MPICH2.

1.4 MOTIVATION

Parallel application developers come from multiple disciplines of scientific research. Their aim is to make their algorithm work more than concerning about improving sustained system performance on HEC machines. This reduces the productivity of these machines and the purpose of building fast supercomputers is defeated. It is necessary to pinpoint the performance bottlenecks and to optimize the performance of parallel codes automatically.

Computer architecture provides various means of improving data access performance, however, it requires locality in data. Many parallel codes access data noncontiguously, where locality does not exist. Instruction level parallelism and memory level parallelism are introduced in superscalar and multicore processors. However, these techniques are beneficial only if data access latency is masked efficiently. Data prefetching has been considered an effective method to mask data access latency. Unfortunately, current data prefetching optimizations are marred by complexity overhead and limiting their effectiveness to a few simple algorithms. There is a need for sophisticated strategies to mask data access latency as well as reducing the divergence gap.

It is the goal of performance optimization techniques to optimize the performance of parallel applications dynamically. These techniques have to avoid any burden on the application developers and utilize numerous structures provided by computer architects.

This dissertation focuses on designs and models that attempt to improve data access performance using automatic loop reordering and data prefetching strategies.

1.5 THESIS

Our thesis is as follows. Existing methods of optimizing data access performance require greater support from compilers and developers, which have been proven ineffective. We seek to provide novel analytical models and architecture to effectively and automatically improve data access performance. Our approach is to predict cache optimization parameters analytically in order to reduce the overhead caused by trial-and-error based methods. We have designed an architecture to prefetch data by using aggressive strategies to predict *what* data an application would use in the future and to push that data closer to processor *in time*. These techniques promise applicability in middleware, and are extendable to various levels of memory hierarchy, such as parallel I/O. The objective of our thesis is to automatically improve data access performance in order to achieve superior productivity on high performance computing machines without placing burden on application developers.

1.6 OVERVIEW OF THE DISSERTATION

This dissertation is organized as follows. Chapter 2 provides a literature review of past memory optimization methods in relation to our thesis. We discuss the multitude of approaches of data access optimization methods and attempt to provide arguments for our prediction models and prefetching strategies. Chapter 3 presents our server-based data push architecture. We discuss existing memory subsystem in multicore processor

architecture and present modifications to the microarchitecture of memory subsystem to monitor data access history and push data from DPS. We also discuss data access pattern prediction in spatial and temporal dimensions. In Chapter 4, we provide fundamentals of data access performance models and discuss our prediction models that can be applied for software level data access reordering. In Chapter 5, we show the practical usage of our prediction model in improving the performance of MPI implementation. Chapter 6 provides the simulation results of our server-based data push model. In Chapter 7, we discuss the applications of server-based push model at various levels of memory hierarchy and ways to find energy savings-memory performance tradeoff. Lastly, in Chapter 8, we present overall conclusions, impact of our work in high-end computing field, and the future directions of this work.

CHAPTER 2

LITERATURE REVIEW

2.1 DATA ACCESS OPTIMIZATION

Poor data access performance has been a growing problem for the last three decades and many researchers attempted to solve the problem. Despite numerous efforts, memory performance is still a bottleneck in high end computing. One major contributing factor to this problem is the growth of processor performance. Processor performance has improved by 52% a year until 2004, and by 25% since [Hepa06]. Memory latencies are high and the number of memory references issued by superscalar processors makes the memory performance even worse. Overlapping the CPU stall time during these memory accesses can be done at hardware level and at software level. At hardware level, providing multiple levels of cache and prefetching data before a CPU requests for it are popular strategies. At software level, modifying applications in order to improve cache utilization and compiler or user inserted prefetching instructions are prominent.

Hardware data prefetching is considered as an effective method of masking CPU stall time. Data prefetching anticipates cache misses and fetches data before processor requests for data. When the anticipation of a future cache miss is correct, CPU stall time that would have been caused due to the cache miss is avoided. The challenge in data prefetching is anticipating future cache misses as well as the time that cache miss occurs. Many researchers have proposed algorithms to predict future references based on the history of cache misses. We discuss existing strategies and their inadequacies in Section 2.2.

Many processor architectures rely on multiple levels of cache memory for exploiting spatial locality and temporal locality of data being accessed. However, this method fails when data accesses are non-contiguous where spatial locality does not hold. Temporal locality does not have effect, when data sizes are very high in addition to non-contiguous data access. Obtaining very high performance is the goal of parallel and distributed programs, where these slow memories prevent in achieving that. The spatial locality of data impacts the performance of parallel algorithms such as the ocean grid solver and Barnes-Hut [Cull97] and other domain decomposition based algorithms. The ocean grid solver exchanges data along horizontal and vertical boundaries. In many domain partition based solutions the boundary data is contiguous and spatial locality is optimal in the cache. When the boundary is non-contiguous (e.g. column boundary in a row-ordered language implementation), the amount of cache misses increase based on the contiguity of the data. The Barnes-Hut application initially operates on adjacent particles with good spatial locality for communication. As the simulation progresses, particles travel through physical space decreasing the spatial locality of communication causing additional cache-related delays. Many scientific applications have similar noncontiguous patterns. Another example, transmission of a sub-matrix may require a series of non-contiguous accesses incurring more memory latency than contiguous accesses of the same size. Transmissions of data in such cases often utilize the message-passing model, a widely used and accepted parallel programming interface called Message Passing Interface (MPI) [Mpif98]. In Section 2.3, we discuss these performance bottlenecks and related work.

2.2 HARDWARE DATA PREFETCHING

Data prefetching is a well studied research area of computer architecture. Traditional hardware data prefetching strategies on single core processors range from simple sequential prefetch strategies to complex Markov prefetching, and to using compiler hints in prefetching and chasing pointers. Sequential strategies [Dads93, Dads95] prefetch next *k* lines of data, while strided strategies [Chba95, Fupa91, Jogr97, Kasi02] predict future strides based on past accesses. With the increasing complexity of these methods, the benefits of prefetching diminish in the traditioanl client-initiated prefetching. Software-controlled prefetching [Mogu91] gives control to developers or compilers to insert prefetching instructions into programs. Many processors provide support for such prefetching instructions in their instruction set. However, software-controlled prefetching puts burden on developers and compilers, and is less effective in reducing memory stall time on ILP processors due to late prefetches and resource contention [Prra99].

With the emergence of multi-thread support in processors, many thread-based solutions have been proposed to deal with the complexity issue. These methods can be roughly classified into two categories: pre-execution based and prediction based. Pre-execution based methods often use a helper thread to run slices of code ahead of main thread. A small list of various proposals using pre-execution include Luk et al.'s Software controlled pre-execution [Lukc01], Liao et al.'s Software-based speculative precomputation [Lwwh02], Zilles et al.'s Speculative slices [Ziso01], Roth et al.'s Data-driven multithreading [Roso01], Annavaram et al.'s Data graph precomputation [Anpd01], and Hassanein et al.'s data forwarding [Hafe04]. Many of these methods often rely on compiler support to select slices of code to pre-execute and to trigger execution of

that code. Collins et al. [Ctws01] suggest using hardware to select instructions for precomputation. Zhou [Zhou05] and Ganusov et al. [Gabu05] proposed utilizing idle cores of a CMP to speed up single threaded programs. Zhou's *dual-core execution* (DCE) approach uses idle core to construct large, distributed instruction window and Ganusov et al's *future execution* (FE) uses idle core to pre-execute future loop iterations using value prediction. In contrast to pre-execution approaches, our Data Push Server (DPS) system resides on a dedicated data server and adaptively chooses future data prediction strategies aggressively. DPS is designed to serve multiple processing cores simultaneously, where as DCE and FE are tightly coupled to one core. DPS predicts temporal patterns to provide *in-time* prefetching, while pre-execution approaches require synchronization to achieve that.

Prediction based multi-threaded strategies use helper threads to predict future references based on history of past accesses. Solihin et al. [Solt02] propose memory-side prefetching (similar to *push-based* prefetching), where a memory processor is designed to reside within main memory. This memory processor observes history L2 cache misses and predicts future accesses. This scheme uses stride-based and pair-based correlations among past L2 cache misses and pushes predicted data to L2 cache. Our DPS strategy suggests using a dedicated server outside main memory to observe data accesses at L1 cache level and to push predicted data directly either into L1 cache or a separate prefetch cache, which is close to CPU. Based on the observed data accesses, DPS has flexibility to choose multiple prediction strategies and to serve multiple processing cores. DPS also predicts *when* to push data based on temporal pattern of data accesses for in-time prefetching. All these features of DPS make a better system than memory-side

prefetching. Hassanein et al. [Hafe04] also use memory-side prefetching and suggest to forward data to either L1 cache or directly into CPU registers, but their scheme is based on pre-execution. Parts of code are run in a memory processor and the predicted data is sent to L1 cache or registers. In this approach, there is a possibility that the memory processor used to run code being not scalable. If many cores of a CPU are requesting the service from this pre-execution based memory processor, the performance of memory processor degrades. In our DPS approach, we use a server outside memory that is more scalable to serve multiple cores of a processor. Since DPS software can be implemented as a thread, multiple threads can be instantiated, where each thread serves an individual core. This makes prediction of future access patterns more scalable.

Furthermore, DPS is extendable to multi-processor environments such as SMP, where nodes share the same memory. DPS fits well as a memory server in these environments. DPS pushes data from the shared memory to local memory of the compute nodes. Since the server-based push model separates data movement from computing, its impact is fundamental and is beyond the field of HEC. For instance, it can serve as the *μ proxy* between the file server and its clients in a distributed file system to improve scalability; can enhance coherence to provide a single image in a parallel system; and can virtualize storage in a Grid environment. Even in HPC, DPS can be enhanced in language, compiler and scheduling, and can be implemented at system or application level.

2.3 SOFTWARE LEVEL OPTIMIZATIONS

Memory performance of non-contiguous data accesses can be improved by various optimization techniques such as array padding, loop un-rolling, loop transformations,

cache blocking etc. Loops are the basic blocks most of the execution time is spent in numerical and scientific applications. Transforming the data access pattern in these is most effective based on the number of iterations each loop runs. *Loop blocking* [W3hp04] minimizes memory system use with multidimensional array elements by completing as many operations as possible on array elements currently in the cache. *Loop unrolling* [W3hp04] attempts to unroll certain innermost loops, minimizing the number of branches and grouping more instructions together to allow efficient overlapped instruction execution (instruction pipelining). The best candidates for loop unrolling are innermost loops with limited control flow. *Loop distribution* [W3hp04] moves instructions from one loop into separate, new loops. This can reduce the amount of memory used during one loop so that the remaining memory may fit in the cache. It can also create improved opportunities for loop blocking. *Loop fusion* combines instructions from two or more adjacent loops that use some of the same memory locations into a single loop. This can avoid the need to load those memory locations into the cache multiple times and improves opportunities for instruction scheduling. *Loop interchange* changes the nesting order of some or all loops. This can minimize the stride of array element access during loop execution and reduce the number of memory accesses needed. *Outer loop unrolling* unrolls the outer loop inside the inner loop under certain conditions to minimize the number of instructions and memory accesses needed. This also improves opportunities for instruction scheduling and scalar replacement. *Data prefetching* [Dkkl99] is an effective technique to hide memory access latency. It works by overlapping time to access a memory location with time to compute as well as time to access other memory locations. This inserts prefetching instructions for selected data accesses. We have to be

careful about choosing these accesses to not to prefetch unnecessary data. This technique works well in combination of loop unrolling.

Some advanced compilers provide these optimizations. There had been an immense amount of research to improve the compilers to include all these optimizations [W3nc00, Dkkl99]. Despite all these efforts, compilers alone could not be very successful due to their compile time, and architectural constraints. For example, cache performance is sensitive to the block sizes of the data and it depends on cache size, cache line size and associativity very highly. Some of the aggressive optimizations may degrade the performance by transforming already optimized code into sub-optimal code.

As an example, we compare compiler optimization performance with manual optimizations to show that the manual optimization is better than compiler could achieve with the best optimization options. We use matrix-transpose (MT) program in this example, executed on SGI Origin 2000 machine with MIPS 10000 processor. We compiled this program with SGI MIPSpro compiler [W3nc00], which has exclusive library of loop optimizations called LNO. For manual optimizations, we used cache blocking and external array padding for array sizes of 512*512, 1024*1024, 2048*2048, 4096*4096, and 8192*8192. Each element is an 8-byte long double. The tested scenarios are:

a) compilation of MT using –O2 optimizations (default) [mO2]
b) compilation of MT using –O2 optimizations + cache blocking [mO2cb]
c) compilation of MT using –Ofast optimizations [mOfast]
d) compilation of MT using –Ofast optimizations + cache blocking [mOfastcb]

It can be seen from Figure 2.1, that default optimizations are worse than simple manual cache blocking optimization. Using advanced compiler optimizations show improvement over the default options. The last column of manual optimizations in combined with compiler optimizations shows another 100% improvement. This shows the scope for further improvement of performance if the code is developed with these optimizations taken into consideration based on hardware capabilities and software requirements in mind.

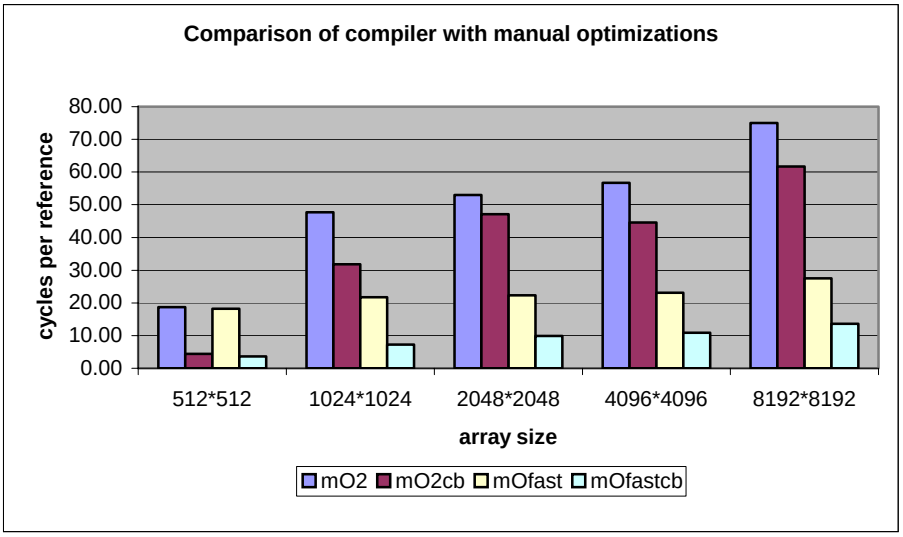


Figure 2.1. Comparison of compiler performance with manual optimizations

Finding an effective combination of these manual optimizations requires extensive knowledge of the hardware architecture and data access patterns of application. If a developer is attempting to achieve high performance, he/she needs to be aware of efficient optimization techniques to be applied in the right place. But in practice, it is not common that application developers are aware of all the advances in technology and it is a costly process to “tweak” the code to achieve better performance. For them, it is more important to make their application work correctly and to meet their deadlines of

application development. It is performance evaluation researchers' job to worry about finding optimizations, without putting burden on application developers.

To solve the above problem, we need better performance analysis tools and optimization suggestion tools. Several trace-based systems have been designed for performance studies of parallel applications, such as Pablo [Derr99], AIMS [Yasm95]. These systems trace performance data for the whole application and analyze that data to find the bottlenecks. If any part of the code is tuned for improvement, the whole application has to be run to see the performance. The Paradyn system [Mcch95] is a dynamic performance instrumentation and measurement system. This system identifies the performance bottlenecks. But there is no mechanism to show the code tuning approaches that help the user or to apply optimizations automatically in order to improve performance. This needs retrieval of parameters of data access pattern at the bottleneck location and prediction of performance with modified patterns. SCALEA is another tool that does instrumentation, measurement, analysis and visualization. But this still lacks finding the effective combination of tuning methods.

Currently there are few automatic tuning software tools. One of the most popular tools of optimization is Automatically Tuned Linear Algebra Software (ATLAS) [Whal01]. This tool runs subroutines multiple times to obtain the best optimization parameters by a trial and error method. Chung et al. [Chuh04] proposes a method to try several optimization values in the valid range, where previous runs define valid range. This uses machine-learning mechanisms to characterize and prioritize performance issues. This method also needs multiple runs to find a good set of optimizations. A prediction model can remove these multiple runs and be extended to optimize more than

just linear algebra subroutines. When the prediction model is simple and fast, it can facilitate to perform the optimization dynamically, at runtime, based on the data access pattern and available memory hierarchy.

Many performance prediction models are available to help programmers in estimating the cost of memory. Copious research effort has been spent in this area to develop accurate cache performance models. But most of these models [Chat00, Sech00, Jaco96] lack generality. They are complex, and are bounded to a few algorithms or data access patterns. Jacob [Jaco96] extracts address traces from the code, which requires execution of the program, and consumes a lot of time if an optimization has to be applied. In our research, we developed a prediction model. We base our prediction model on various access patterns, which are parameterized. This helps in predicting the memory cost with very small complexity and skips the costly process of tracing the references every time data access pattern is changed. Chatterjee et al. [Chat01] studies the exact analysis of cache misses based on the polyhedral model, which is very complex. The Cache Miss Equations model (CME) [Ghos99] is the least costly performance model to our knowledge. However, this model also requires tracing the references to create the reuse vectors and solve cache miss equations. These models are accurate but expensive, and are better choices for static analysis of cache behavior. Our model fits better in choosing the optimization parameters dynamically at runtime than CMEs.

Our model focuses on a wide range of data access patterns with multiple array variables. Most of the other cache analysis models hold good results for a specific algorithm [Chat00, Sech00], but fall short in acquiring generality. In our research, we

aimed to develop models that are simple, practical and reasonably accurate to make performance optimization decisions.

2.4 SUMMARY

In this chapter, we presented research work related to data access performance optimization at hardware level and software level. At hardware level, data prefetching is an extensively studied area. Various strategies exist to anticipate future cache misses and data access patterns. At software level, advanced compilers perform loop optimizations; however, they are ineffective with many complex noncontiguous data access patterns. We also discussed memory optimization techniques performed by various frameworks. Despite these efforts, a huge gap between peak performance and sustained performance with numerous applications still exists. Many data prefetching methods need to be more accurate in predicting future reference by using sophisticated and adaptive algorithms. Most of the software level optimization methods are either time consuming or not useful for generic applications. Our research goal at hardware level is to utilize a server and to push data in time to mask the data access latency effectively. At software level, our objective is to provide models that can be used for any application and they are based on data access patterns that cover a wide spectrum of memory accesses. The following chapters provide detailed discussions of our research towards the goal of data access performance improvement. We discuss more related work depending on the context, when we introduce our strategies, models, and implementations.

CHAPTER 3

SERVER-BASED DATA PUSH ARCHITECTURE

In this chapter, we first introduce the operation of memory hierarchy, memory subsystem of existing processors, and the prefetching support in current general-purpose processors. We then present the design and functionality of our server-based data push architecture. We discuss various technical issues, including how to monitor data access history, what data to prefetch, when to prefetch, how to push data. Finally, we discuss the benefits of implementing DPS on a multicore processor and present a case study of implementing DPS in Cell processor.

3.1 MEMORY SUBSYSTEM

To bridge the gap between processor and memory performance, modern computer architectures include multiple levels of memory hierarchies. Cache memory and Translation Look-aside Buffer (TLB) are essential parts of this hierarchy. Cache memories are placed either on the die of a processor or outside between CPU and main memory. These are introduced to hold small portions of the contents of main memory that are (believed to be) currently in use. The size as well as the latency increases to access a level of cache as it is placed further from the CPU. It is common to place L1 cache on the die of the processor to make it closer to the CPU. Recent processors (Intel Core Duo architecture [Dowe06] based processors, IBM Power5 and AMD Dual core processors) place L2 cache on the die and extending the cache to another level L3. AMD's processors use large L2 cache instead of L3 cache. In Intel and AMD's dual-core processors, each core has its private L1 cache and shared L2 cache. Cache of virtual

memory mappings is stored on Translation Look-aside Buffer (TLB). Processors use virtual addresses to refer to data. These addresses have to be mapped to physical addresses on main memory. Page table contains the full list of such virtual address to physical address mappings and a TLB acts as a cache to store a small number of recently used mapping information. It is becoming common to include more than one level of TLB. IBM's 64-bit PowerPC processors use Segmentation Look-aside Buffer (SLB) to cache the segment information. This replaces segment registers of previous 32-bit architectures of PowerPC.

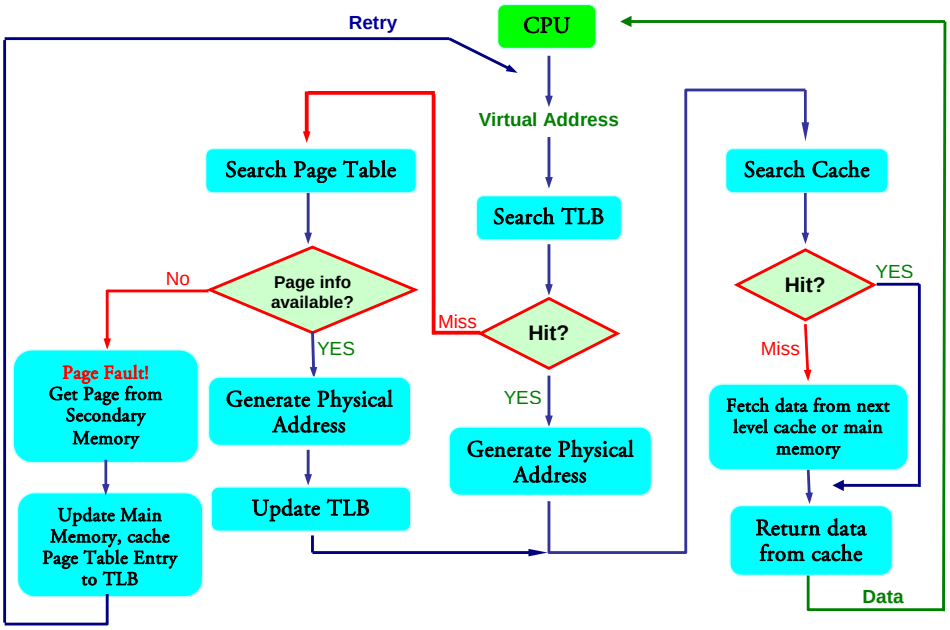


Figure 3.1. Processing data request by CPU

Figure 3.1 explains how CPU processes a memory request on a traditional single core processor with one level cache and a TLB. We assume that the memory hierarchy follows inclusive property in this subsystem. Baer et al. [Baer88] suggests that it is essential that the contents of higher level cache be a subset of the lower level caches, so that the higher level caches shield the lower level caches from cache coherence interference. Most of current memory hierarchies follow this property. When a memory reference occurs, CPU

initially searches for data in the registers (not shown in the figure). Registers are small in number at the highest level of memory hierarchy and the fastest accessible. If the data is not present in the registers, CPU searches the TLB to find the virtual-to-physical address map. If the mapping entry is found, the next step is to search the L1 cache for the data. If L1 cache does not have the data (which results in an L1 miss), the search continues to the next level cache. If the requested data was never fetched from the main memory, cache misses occur at each level cache. When data is found it is fetched and stored in the L1 cache and returned to the CPU. In the figure, we have shown only one level of cache. Some of the advanced processors search TLB and the L1 cache simultaneously, where cache maintains the virtual address to real address mapping. For the purpose of simple explanation, we assume that the search is sequential. If a virtual-to-physical address mapping is not found in the TLB, the search continues to find the mapping in page table, which resides in main memory. If a requested page has never been fetched into the main memory, the mapping of that page does not exist in the page table, which results in a page fault. In this case, the requested data page is fetched to the main memory and the page table entry is updated. The total cost of accessing memory includes the access time and the miss penalties of these levels in the hierarchy.

The recent revolution of multi-core (dual-core, quad-core and eight-core) processor architectures have increased the computing capacity rapidly. Single core processor speeds have been following the Moore's law, which resulted in 52% improvement in speeds till 2004. From then, Intel, AMD, IBM, Sun has introduced multicore processors, with various designs based on Chip Multi-processing (CMP), thread level parallelism (TLP). In these processors, each chip has more than one core that contains Arithmetic Logical

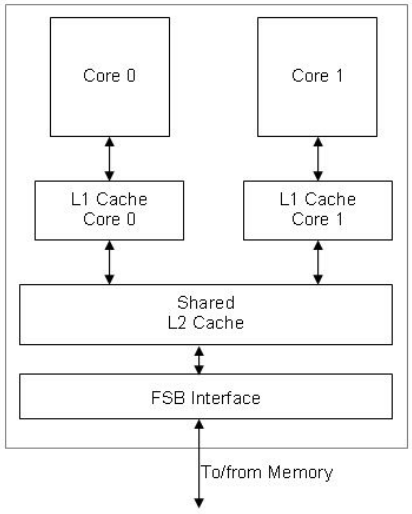


Figure 3.2. Multicore processor architecture (Intel Core)

Unit (ALU) and one or two levels of cache memory. A typical Intel multicore processor [Dowe06] is shown in Figure 3.2. In this processor, each of two cores has its own L1 D-cache and shared L2 D-cache. The data access operation by CPU cores is similar to the operation shown above (Figure 3.1), except sharing the L2 cache. L1 misses from both cores access shared L2 cache memory. IBM Power5 dual-core processor has similar cache hierarchy with shared L2 and L3 cache memories [Skte05]. Sony/Toshiba/IBM's Cell Broadband Engine (CBE or Cell) is a heterogeneous architecture. Cell processor contains eight symmetric processing elements (SPE core) and one PowerPC processing element (PPE). Each SPE has access to a small local store (local memory) and share a system memory. This multi-core processor revolution has given us an opportunity to introduce new features such as separating the task of data access to improve data access performance.

Figure 3.3 shows the detailed memory subsystem of Intel Core architecture [Dowe06]. Schedulers (Reservation Stations) issue various instructions including memory operations. Load and store instructions go to Memory Reorder Buffer (MOB),

where the MOB schedules memory instructions to improve memory level parallelism. If the requested data is in the L1 cache, data is moved to ALUs for processing. If data is not present in L1 cache, a cache miss occurs and propagates to L2 cache and so on. DTLB acts as a cache for memory mapping translations between virtual and physical addresses.

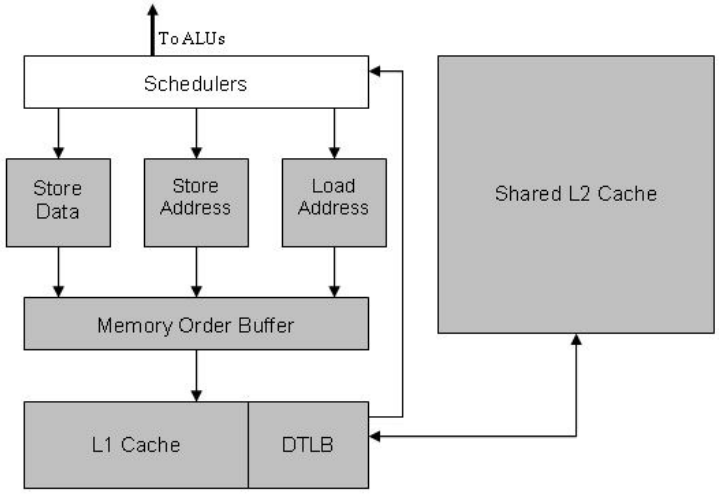


Figure 3.3. Memory Subsystem of Intel Core architecture

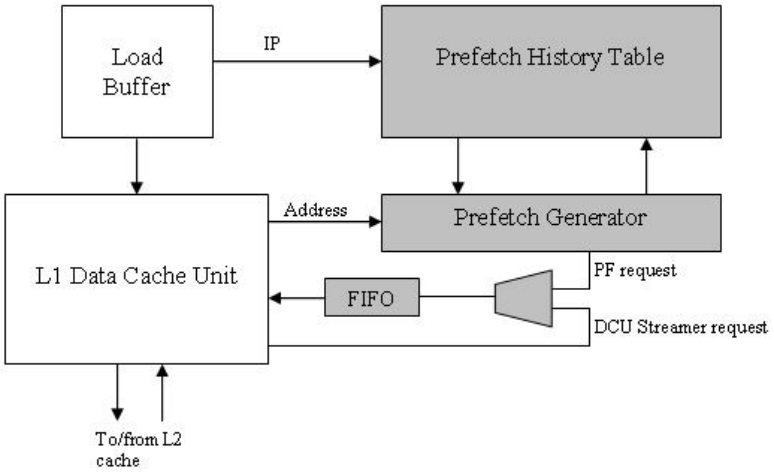


Figure 3.4. Instruction Pointer-based prefetcher of Intel Core architecture

The Intel Core microarchitecture provides Smart Memory Access (SMA) system [Dowe06], where Instruction Pointer-based (IP) prefetcher is added to L1 level cache (see

Figure 3.4). This prefetcher predicts memory addresses that are going to be used by a program and deliver data just in time. IP prefetcher maintains a Prefetch History Table, where it stores the last address, last stride and last prefetch information. However, the IP prefetcher only “tries” to predict the address of next load, according to a constant stride calculation. When a constant stride is found, Prefetch Generator issues a prefetch request to L1 cache. To avoid contention between regular instructions and prefetch instructions for bandwidth, prefetch requests are given lower priority. If the prefetched data arrives late or predicted references are dropped due to lower priority, the benefit of prefetching is lost.

As mentioned in the previous chapters, the existing prefetching solutions are limited by the power of prefetch generator. They often try to predict only the next load instruction based on constant stride. With the emerging multicore processors with numerous processing cores, not all the cores would be busy with processing. Using the computing power of idle cores or dedicated cores for making data access prefetching decisions is an obvious solution to data access problem. We propose to utilize the power of these computing cores to execute better algorithms to predict complex data access patterns that can adapt the prefetch distance based on latency. The current prefetchers support client initiated *pull-based* prefetching. We aim to use *push-based* prefetching that effectively separates data access from computing as suggested by Smith [Smit82].

3.2 DATA PUSH SERVER ARCHITECTURE

Data Push Server (DPS) is designed to predict data access patterns of applications and to push the predicted data from main memory to a cache closer to processor. Figure 3.5

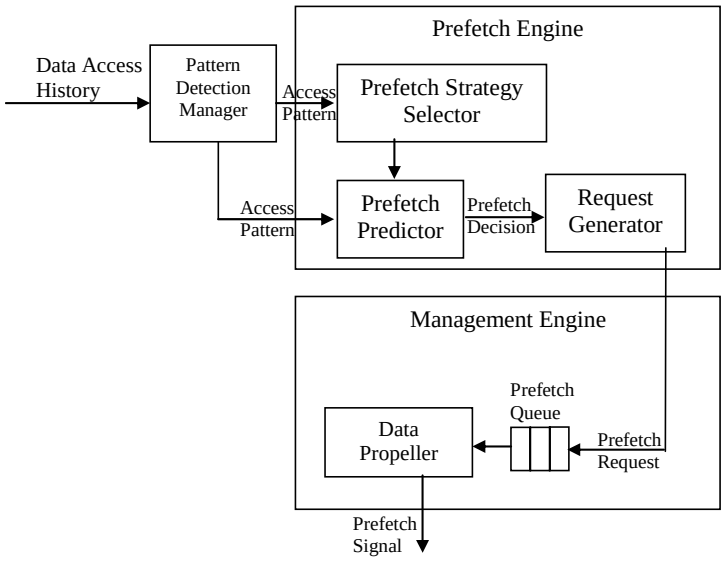


Figure 3.5. Components of Data Push Server

shows the structure of DPS. Its three primary components are: pattern detection manager, prefetch engine, and management engine. The *pattern detection manager* (PDM) collects history of data accesses in spatial and temporal dimensions. Data access information in spatial dimension includes the strides between successive accesses. Information in temporal dimension refers to the time of accesses, either in clock cycles or inter-reference distance. The PDM then classifies patterns of those data accesses. The prefetch engine is responsible to predict future accesses and the timing. It in turn has three subcomponents: prefetch strategy selector, prefetch predictor, and request generator. The *prefetch Strategy Selector* (PSS) adaptively selects an appropriate method to predict future accesses based on the pattern information. The *prefetch predictor* of the prefetch engine decides *what* data to fetch and the *request generator* decides *when* to push data so that the prefetched data arrives at its destination *in time*. Here by '*in time*', we mean that data is pushed from its source to destination within a window of time before it is required, and where it does not replace other data blocks from cache falsely. By moving data into a cache too early, it may replace data blocks that would be accessed in the near future. Our strategy aims to

avoid such negative effects. The Management engine is responsible to issue instructions to push data. The prefetch requests are kept in a prefetch queue and *data propeller* in the management engine issues a signal to push the data to its destination. The source of data in multi-core processor environment is main memory, and the destination is cache memory. Also, when prefetching fails, management engine holds the cache misses as usual.

Figure 3.6 shows a scenario of DPS system running on a computing core, serving

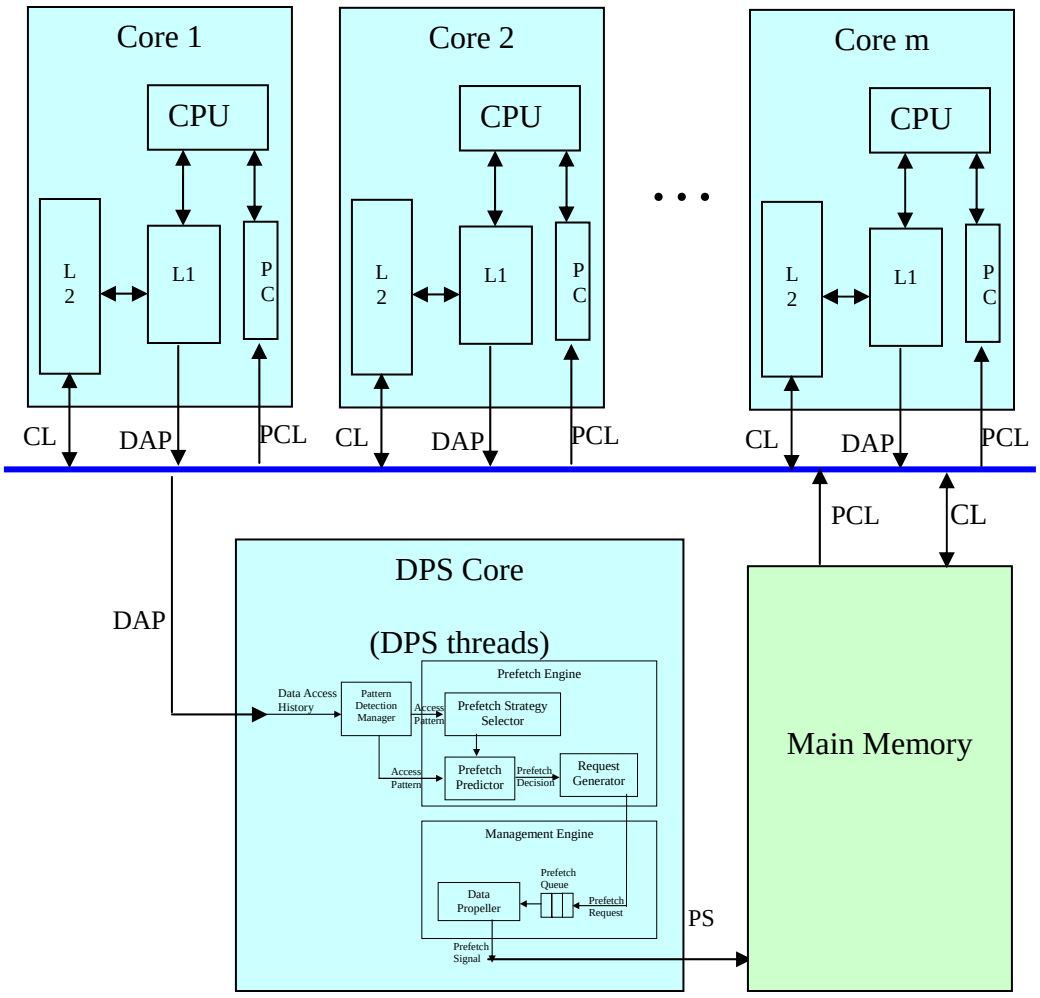


Figure 3.6. DPS on Multicore processor
(DAP: Data Access Pattern, PS: Prefetch Signal, CL: Cache Line,
PCL: Prefetched Cache Line)

processing cores 1,2, …, m. We show that each core in a multicore processor environment contains its own L1 and L2 cache memories and shares the memory among other cores. This assumption is true for the future multicore processors announced by Intel, AMD and IBM. The core, on which DPS is running, observes the data access patterns of L1 cache of cores 1 to m, and predicts the future accesses correspondingly. The data (prefetched cache line or PCL) is pushed from the shared main memory to the prefetch cache (PC) of each client core. Data Propeller issues the prefetch signals (PS) to main memory and data is pushed. Regular memory operations related to raw cache misses caused by an application are served by main memory directly. These cache lines are read or written by L2 cache and this data (CL) is transferred main memory and L2 cache. CPU on each core accesses both L1 cache and prefetch cache simultaneously. An L1 cache miss is propagated to lower level L2 cache. A prefetch cache miss is discarded. In the following sections, we discuss the functionality of DPS system components in detail.

3.3 MONITORING DATA ACCESSES

Monitoring data access history is an important task in order to predict future accesses. There are various methods to obtain an insight into future data accesses by an application, such as user provided hints, compiler hints, and predicting based on history of accesses. If an application developer has prior knowledge of data access patterns in his application, that information can be used. Another method is to utilize compiler provided hints. Existing advanced compilers perform extensive data flow analysis during compilation. Some of this data flow analysis is used to generate software level prefetching instructions

[Modk96]. The most common data access monitoring method is to observe the past history to predict sequences of future addresses. Similar to Intel Core architecture shown in Figure 3.4., many existing processors use past history of data accesses. Many researchers [Dads95, Vali00] also consider observing data access history an economical and productive way of predicting future accesses due to minimal hardware modifications.

The issue of where to observe the data accesses comes next. Data access history can be observed from cache accesses at L1 cache, cache misses at L1 cache, cache misses at lower level caches. Many previous studies use cache miss history only either at L1 cache or at L2 cache. This can predict future accesses to some extent; however, the accuracy of prediction is low for complex patterns. Moreover, observing cache misses only cannot give an insight into data access times, which is necessary to predict *when* to prefetching. In our technique, DPS observes the data access history at L1 level cache. DPS observes access history of both hits and misses, which helps us in predicting time of future accesses. As shown in Figure 3.7, DPS obtains the history of data accesses at L1 cache to construct comprehensive history, which improves prediction accuracy.

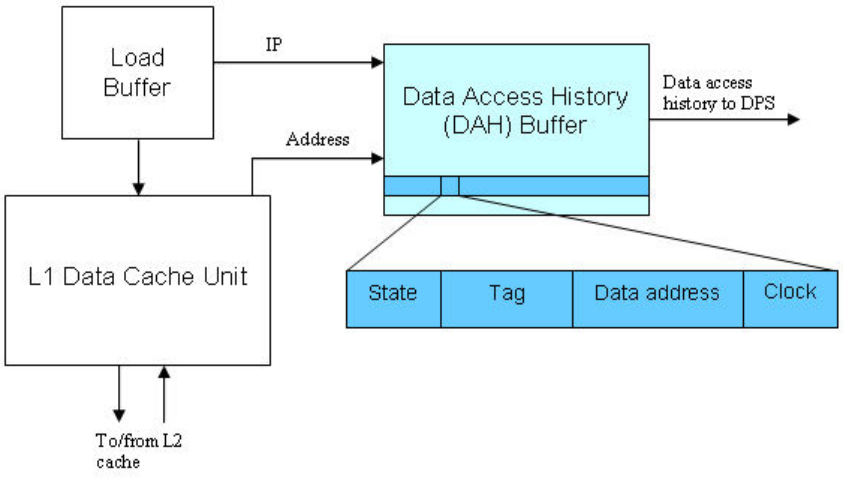


Figure 3.7. Data Access History collection for DPS

The history of data accesses contains state, tag, data address, and clock cycle information. State refers to L1 cache access or L1 miss. Tag field records the instruction address and data address refers to physical address of data requested by the CPU. Clock field records the time of access in cycles. This information is used to predict the temporal pattern of the data accesses and to push data *in time*. As the clock through out the CPU is similar, the core on which DPS is running on has the same clock to synchronize *in time* prefetching.

The data access history is collected temporarily into DAH buffer (Figure 3.7). This buffer is accessed by DPS system. We propose a new instruction called `GDAH` to retrieve the contents of DAH buffer. `GDAH CORE_ID` instruction retrieves data access history from a core identified by `CORE_ID`. This instruction provides flexibility of implementing DPS either at hardware level of system level. At system level, DPS can be implemented using helper threads on one of the cores. This is a scalable solution as multiple threads are able to support multiple processing cores.

After collecting history of data accesses, prediction of future accesses is done by the *prefetching engine* of DPS. In research literature, there are many strategies to predict future data references. However, no single strategy accurately predicts all data access patterns. Sequential and strided strategies can predict regular constant and varying strided accesses, while another set of strategies try to chase pointers and data structure traversals [Anpd01, Kcky01, Roms98] that require compiler and user provided hints. Pre-execution based approaches [Gabu05, Hafe04] often use a helper thread to run slices of code to predict future accesses. Complexity of these strategies varies. Using simple strategies cannot capture complex patterns and complex strategies suffer from high overhead in

predicting simple access patterns. An accurate prefetching mechanism should support various prediction strategies and should adapt to data access patterns of an application at runtime.

In DPS, the *pattern detection manager* (PDM) detects data access patterns, and the *prefetch strategy selector* chooses an appropriate prediction strategy based on the detected pattern. To detect whether a pattern is formed by simple strides or complex variable strides, the PDM observes the distances (spatial and temporal strides) between consecutive data references. We classify data access references into contiguous, non-contiguous, and combinations of contiguous and non-contiguous patterns. Figure 3.8 shows a classification data access patterns. We divide these patterns further based on repetition of occurrence of each pattern and on variation of strides for non-contiguous patterns. Each pattern is divided into single occurrence, and repetitive patterns. For single occurrence reference pattern, if there is no regularity, future accesses cannot be predicted. Among repetitive patterns, even if there is no regularity in a sequence of accesses, future accesses can be predicted using the past sequence of accesses.

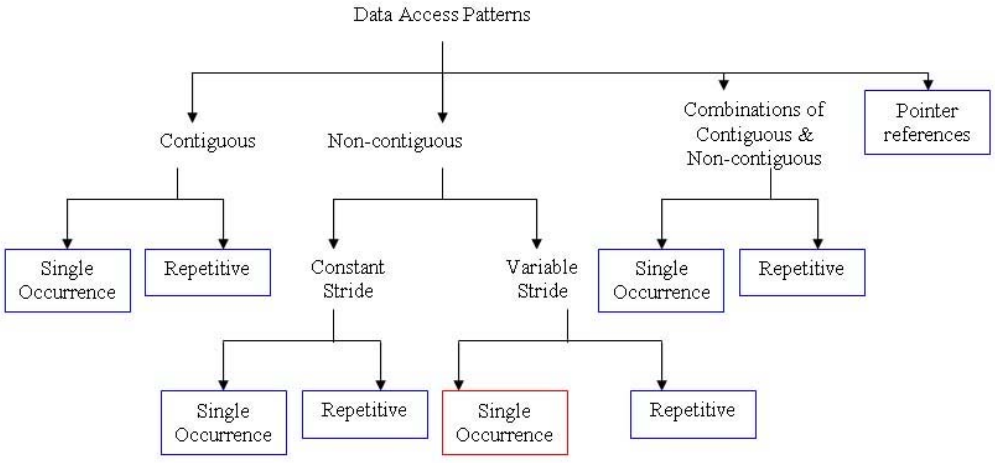


Figure 3.8. Classification of Data Access Patterns

Based on above classification, the PDM characterizes a pattern and passes that information to the *prefetch strategy selector* (PSS). The *prefetch strategy selector* (PSS) chooses a prediction strategy based on initial information regarding a pattern. Many strategies exist to predict future references with similar strides or patterns of strides [Chba95, Fupa91, Kasi02]. Sequential prediction simply adds 1 to current address and brings the next block of data. This is simple and useful to predict contiguous access patterns. Strided prediction uses differences between successive data accesses. If a constant stride exists, future accesses are calculated using that stride. For example, if *r* is the current data access and *d* is the constant stride, future accesses are, *r+d*, *r+2*d*, *r+3*d*, …, *r+n*d*, where *n* is the prefetch distance. Stride prediction is useful for constant noncontiguous data access patterns. Using Markov chains [Jogr97] is a more complex strategy, where each data access reference is given some probability of occurrence based on its history. Distance prefetching [Kasi02] uses similar approach, but counts the occurrence of distances between successive references. These two Markov chain based strategies are useful for repetitive patterns.

Table 3.1 gives a summery of prediction strategies that can be used for various data access patterns used in the classification shown above. Sequential, strided and Markov chain based methods predict work well for constant strided patterns, but they are only effective partially when repetitive patterns. Complex variable strided patterns, combinations of contiguous and noncontiguous patterns, and pointer references (marked as XXXX for prediction strategy in the table) need aggressive algorithms in predicting future references accurately. In the next section, we introduce a more complex strategy for predicting future strides.

Table 3.1. Prediction strategies for data access patterns

Pattern	Example of stride sequences	Prediction strategy
Contiguous, Single occurrence	1,2,3,4,5…	Sequential prediction
Contiguous, repetitive occurrence	1,2,3,4,5…, 1,2,3,4,5…	Sequential and Markov chain prediction
Noncontiguous, constant stride, single occurrence	1,5,9,11,…	Strided prediction
Noncontiguous, constant stride, repetitive occurrence	1,5,9,11,…, 1,5,9,11…	Strided prediction, Markov chain prediction
Noncontiguous, constant stride, repetitive stride occurrence	1,5,9,11,…, 2,6,10,12…	Strided prediction, distance prediction
Noncontiguous, variable stride, single occurrence	1,3,9,13,…	XXXX
Noncontiguous, variable stride, repetitive occurrence	1,3,9,13,…, 1,3,9,13	Markov chain prediction, distance prediction
Combinations of contiguous and noncontiguous, single occurrence	1,2,3,9,15,21, 22, 23, 24, 30…	XXXX
Combinations of contiguous and noncontiguous, repetitive occurrence	1,2,3,9,15,21, 22, 23, 24, 30, 1,2, 3, 9…	XXXX
Pointer references	No specific pattern	XXXX

3.4 PREDICTION OF SPATIAL ACCESS PATTERNS

As mentioned in the previous section, many strategies exist to predict future references. However, patterns with variable strides and repetitions need more analysis to find regularity among them. With dedicated machine, as computing power is available for prediction, we introduce a new method that predicts regular patterns with constant stride

as well as variable stride accesses and repeating patterns. This prediction strategy is based on a finding *"what number comes next"* in the context of number sequences [Cogu96]. This method forms a *difference table* of depth *d*, which we call *multi-level difference table (MLDT)*. Existing strided prefetching [Chba95, Fupa91] and distance prefetching [Kasi02] methods use the distances (strides) between successive page numbers up to one level to find regularity. In MLDT scheme, we extend finding distances for more than one level. Each entry of the *difference table* is the difference between the two entries just above it (in the sense "right entry minus left entry").

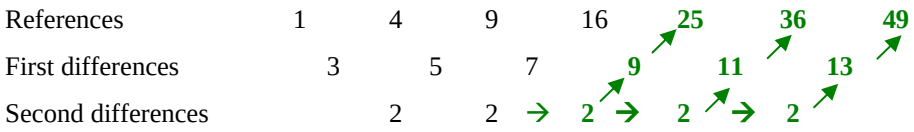


Figure 3.9. Multi-level Difference Table for variable stride non-contiguous pattern

Assume that successive data references are in the order shown in the first line of Figure 3.9. The first differences (d=1) are the strides between the right reference minus the left reference. If these strides are different, differences among these strides are calculated. Second differences (d=2) in Figure 3.9 are equal to a constant value of 2. After a constant difference is found, the next entry of second differences above can be predicted to be the same. As shown in figure 3.9, third entry of second differences is predicted as 2 (predicted entries are marked in bold face and green color font). This is added to the third entry of the first differences, i.e. 7 + 2 = 9. This value is added to the fourth entry of references to find the fifth reference, i.e. 16 + 9 = 25. The future references are predicted (36, 49) in the above example.

In the example above, we have shown finding the address of next data block. We have worked out to find polynomials to predict next k^{th} reference in a reference sequence.

In figure 3.10, we show a three-level difference table. The references are represented by A_i (i = 0 to n). The first differences are B_i, (i = 0 to n-1), second differences are C_i, (i = 0 to n-2), and third differences are D_i, (i = 0 to n-3). We present these polynomials up to the depth of three, which can be extended further.

n	0	1	2	3	4	5	6
References	A_0	A_1	A_2	A_3	A_4	A_5	A_6
First differences		B_0	B_1	B_2	B_3	B_4	B_5
Second differences			C_0	C_1	C_2	C_3	C_4
Third differences				D_0	D_1	D_2	D_3

Figure 3.10. An Example of Multi-level Difference Table

If the depth of a difference table is 1 and a constant value can be found among the first differences, i.e. in figure 3.10, B_i (where i = 0 to n-1) is the constant B, value of k^{th} reference from reference A_r can be found with the following formula. $A_{r+k} = A_r + k * B$.

For a difference table of depth 2, where second differences are constant, value of k^{th} reference from reference A_r can be found with the following formula.

$$A_{r+k} = A_r + k * B_{r-1} + \frac{k*(k+1)}{2} * C$$

For a difference table of depth 3, where third differences are constant, value of k^{th} reference from reference A_r is: $A_{r+k} = A_r + k * B_{r-1} + \frac{k*(k+1)}{2} * C_{r-2} + M_k D$.

Here $M_k = \frac{k}{6} * (k-1) * (k-2) + k^2$, where k = 1, 2…

MLDT works similar to the existing stride based prefetching strategies that predict constant stride patterns. In addition, this method finds sets of repeating differences, and ultimately finds the actual pattern in accessing structures with variable strides data access pattern. When the depth of a difference table is 1, references have a constant stride.

Existing stride-based prefetching strategies are effective for this special case. For variable strided patterns, MLDT searches for regularity among data references by finding deeper difference table. We extend this method to find repeating sets of strides (e.g. 4, 8, 4, 4, 8, 4, 4, 8, 4…) at each level of difference table. These repeating sets of patterns are common in accessing user-defined datatypes (e.g. structures). For patterns with various strides among multiple variables, the prefetch predictor requires more time to learn data access patterns. This can be extended to time series analysis such as ARIMA models [Bogr94] to make complex pattern predictions.

Identifying patterns in irregular accesses (for example pointer references) is complex, which may be impossible to predict the addresses of future accesses. The overhead during pattern learning phase of such data accesses can be reduced by utilizing compiler hints and application support [Kcky01, Roms98]. Most of current compilers perform highly sophisticated data dependence analysis. From this analysis, compilers can generate hints to derive data access patterns. Similarly, the application developers can provide hints regarding the data access pattern in their applications. Runtime profiling of the application also provides such hints. DPS can dynamically select multiple strategies based on the patterns and compiler-provided hints. DPS system is scalable to add new prediction algorithms as it can be implemented at system level and be separated from computing.

3.5 PREDICTION OF TEMPORAL ACCESS PATTERNS

The issue of *when* to prefetch in the existing methods is limited by the occurrence of an event such as a cache miss or a page fault (prefetch on miss) or the first access to a

data block (tagged prefetch) etc. However, these strategies do not guarantee that the prefetched data will reach its intended destination "*in time*" to overlap the processor stall time. The efficiency of prefetching in time depends on three factors (Figure 3.11): the time to predict future accesses (T_{pred}), the latency of initiating and transferring data from its source to destination (T_{lat}), and the gap between current time and the next data reference that would cause a demand cache miss (T_{Δ}) when no prefetching is applied. The prediction cost T_{pred} includes the cost for validating prefetches to avoid duplicate data. Sometimes, it is possible that cache lines are already available in cache memories. Validation avoids such duplication to maintain coherency of data. If T_{miss} denotes the penalty caused by a cache miss, prefetching can take place in the following situations:

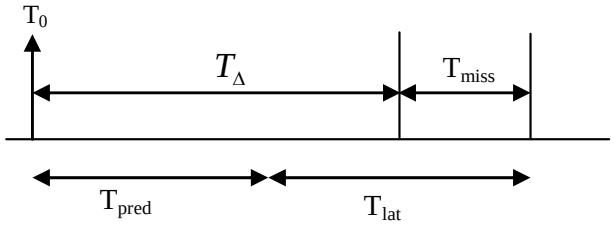


Figure 3.11. In time prefetching

- **Case 1**. If (T_{pred} + T_{lat}) > (T_{Δ} + T_{miss}), the prefetching is completely useless.
- **Case 2**. If (T_{pred} + T_{lat}) > T_{Δ} and (T_{pred} + T_{lat}) < (T_{Δ} + T_{miss}), there is a partial gain of performance improvement based on how much of T_{miss} is overlapped.
- **Case 3**. If (T_{pred} + T_{lat}) = T_{Δ}, the prefetching is *in time* and the prefetching is the most effective (Figure 3.11).
- **Case 4**. If (T_{pred} + T_{lat}) < T_{Δ}, there are two cases.

A. If the destination of prefetched data has empty space to accommodate, the prefetching has no negative effect.

B. If the destination of prefetched data is full, a victim cache line has to be replaced based on replacement policies. This negative effect of prefetching may result in extra cache misses.

To benefit from prefetching, a prefetching strategy has to be adaptive to decide if a prefetch would be useful or not. A useless prefetch increases traffic of the bus, and may pollute a location on the destination of that prefetch. This necessitates the prediction of T_{Δ} to make a decision whether to prefetch or not.

In DPS (Figure 3.5), the *request generator* decides when to prefetch. The request generator varies the value of *k* (from section 3.4) based on the detected spatial and temporal data access history of a cache. Temporal history contains clock ticks of processing core to recognize its timing pattern. The request generator predicts T_{Δ} and adjusts the value of *k* so that ($T_{pred} + T_{lat}$) is approximately equal to T_{Δ}. We assume that only one application runs on a processing core at a time, since it is complex to observe temporal pattern of data accesses when multiple tasks are running on the same core. We currently use MLDT method to identify temporal pattern in order to predict T_{Δ}. This can be extended to use ARIMA models [Bogr94] to predict temporal access patterns.

3.6 PUSHING PREDICTED DATA

The *data propellor* component of DPS delivers data to processing units. After predicting the addresses of future references by the prefetch engine, the data at these addresses has to be delivered to appropriate processing units. In traditional hardware

prefetching strategies, prefetching instructions are issued by the same processing unit that executes a program. In DPS strategy, the predicted future data references are stored in a prefetch queue. The prefetch engine sends this prefetch queue to the data propellor, and the data propellor issues prefetching (push) instructions to move the data from the memory to processing units that need data.

Special support to issue push instructions to memory is needed. In addition to current load and store instructions, we suggest to introdice a new instruction called `PUSH`, which takes destination (client of DPS) core identification, and predicted address of data. The instruction looks as follows:

```
PUSH CORE_ID, ADDR
```

`PUSH` instruction loads the data at address `ADDR` into prefetch cache of core identified by `CORE_ID`. This instruction initiates a search for address `ADDR` at L2 cache, but data is pushed into the prefetch cache instead of L1 cache. Motivation behind using a separate prefetch cache is discussed in the next paragraph. If an L2 cache miss is resulted in due to this instruction, data is brought from memory is directly pushed into the prefetch cache.

Figure 3.12 shows the modified memory subsystem to support DPS prefetching. Data can be directly pushed either into L1 cache or a separate prefetch cache. However, to avoid competition between regular data fetches and data prefetch for L1-L2 bus bandwidth, we propose to use a separate prefetch cache. The L1 cache is usually busy serving loads and stores from CPU. Adding another operation of pushing data might clog the L1 cache. One way to resolve this issue is to give low priority to storing prefetched data. However, the benefit of prefetching data is reduced if it is performed with low

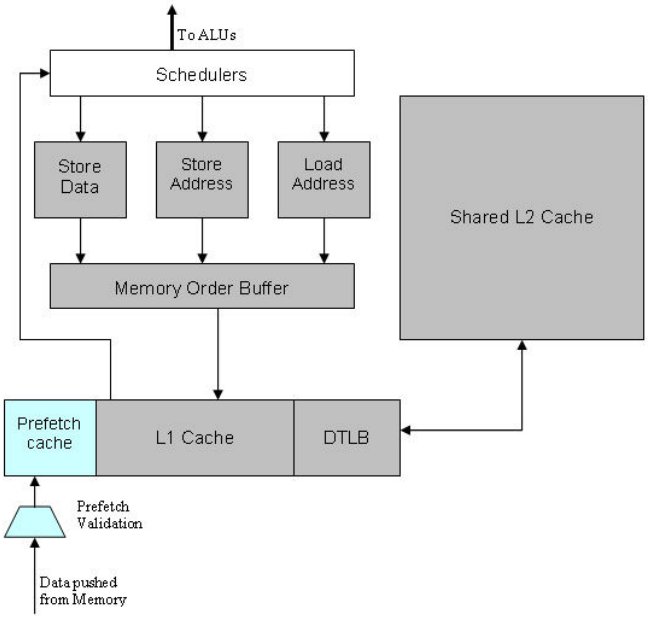


Figure 3.12. Microarchitecture of memory subsystem for DPS

priority. Using a separate prefetch cache, which can be searched simultaneously with L1 cache solves improves prefetching benefits. The proposed prefetch cache is a small cache, which is accessed simultaneously with L1 cache. The size of cache line for the prefetch cache is similar to that of L1 cache. We suggest to use higher associativity for the prefetch cache, as addresses in regular patterns have affinity, and they tend to map to the same location in the prefetch cache. But care must be taken that access time for prefetch cache be less than that of L1 cache.

In many current prefetching schemes, data is directly pulled (prefetched) into L1 or L2 cache memories. This avoids duplicate cache lines. In our prefetching strategy, DPS pushes data into prefetch cache directly. In this case, care must be taken to avoid duplicate copies of cache lines. This is important especially when the data has already been modified and kept either in L1 or L2 cache. To solve this problem, we use a prefetch validation component. The validation component searches the L1 and L2 cache

if prefetched lines are already available in caches. It discards prefetched lines if they exist in cache memory. DPS alleviates duplicate addresses by monitoring hits and misses of L1 cache as mentioned in section 3.3. The operation of prefetch validation avoids the possibility of duplicate cache lines further and manages cache coherence. The performance of prefetching would not be affected as this validation cost is small and is usually constant (equal to average cache hit time). The request generator of DPS considers the validation cost and adapts to generating prefetching request further in future.

The prefetch cache contains the physical addresses. It is searched for physical address similar to the search operation of L1 cache. For memory mapping from virtual to physical address, TLB entries are used. This avoids complex virtual prefetch cache, which contains address mapping information [Hepa06]. If a mapping is not found, a TLB miss occurs. As the size of TLBs in current processors is big enough, it is safe to assume that the number of TLB misses are few as it can hold the mappings for much of the working set. In our simulations, we observed that with the existing TLB configuration of 128 to 256 entries, it can hold mappings for a large working set. For instance, with 4 KB page size, 256 entry TLB can hold up to 1 GB working set. The TLB misses we observed for various SPEC 2000 benchmarks are few and related to cold start misses.

The operation of CPU's prcessing a memory request is modified with the addition of prefetch cache. Figure 3.13 shows the modifications from Figure 3.1. DPS pushes predicted data into prefetch cache, which is validated if it is already in the cache memory. Duplicate copies are discarded, which is especially required for dirty cache lines to maintain coherence. From CPU's memory requestion, after virtual to physical address

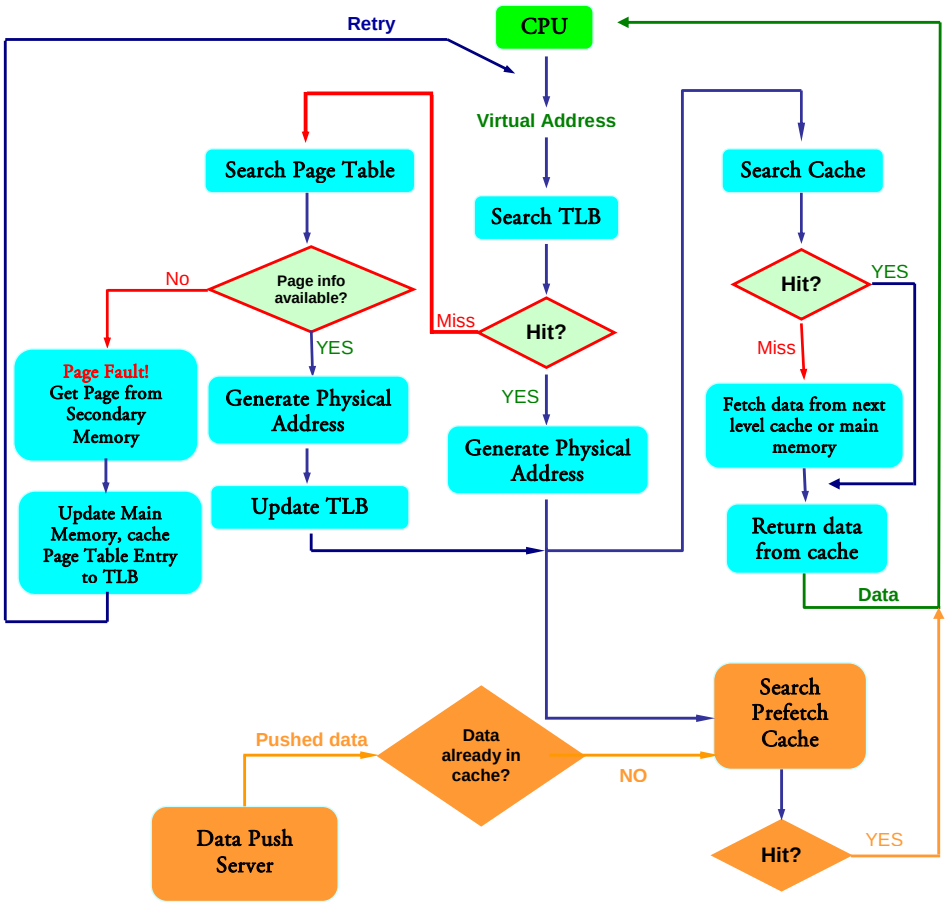


Figure 3.13. Modified CPU data request operation

mapping from TLB is obtained, L1 cache and prefetch cache are searched simultaneously. If the requested cache is found in the prefetch cache, data is sent to CPU directly and cache search operation is discarded. A prefetch miss does not trigger any events. If data is not found both in prefetch cache and L1 cache, then an L1 cache miss occurs and the next level cache or memory is searched for data, similar to normal data access operation. As mentioned in the previous paragraph, no extra TLB prefetching is perfomed to reduce the complexity. If a TLB miss occors, mapping information is brought into TLB from page table and search for cache line resumes by searching prefetch cache and L1 cache.

3.7 BENEFITS OF DPS

Design of Data Push Server has several benefits. It provides true separation of data access from computing. Complexity of data access prediction is moved to a server. This feature gives a chance to select prediction algorithms adaptively based on history. Its data push approach improves performance more than pull-based strategies. The usage of a server also makes the DPS prefetching system scalable. One DPS system supports multiple cores by starting threads for each client.

Separation of data access from processing is a major benefit of DPS. Previous research efforts have achieved such separation to some extent by using helper threads to pre-execute parts of the code [Lukc01, Lwwh02, Ziso01, Roso01, Hafe04] or by using memory processor [Solt02, Hafe04]. Pre-execution strategy suffers from synchronization costs and requirements of compiler or user support to start and stop pre-execution. Memory processor method can suffer scalability issues, when multiple processing cores requests for service as it is bound to inside main memory. More processing power cannot be added or reduced based on demand of client cores. Our DPS system crosses these two major hurdles in separating data access from computing. DPS uses a data access history buffer support to observe data access patterns and pushes the predicted data into a prefetch cache in a timely manner. This does not exclusively require compiler or user support to identify data access patterns. No special synchronization is needed between processing cores and the core, where DPS is running on. This provides true separation of data access from computing.

DPS is a true *push-based* prefetching strategy, which has more potential to have performance gains. Conventional *pull-based* approaches often suffer to run complex

prediction algorithms. In single core processors, prefetching strategies have been given lower priority compared to main processing as the complexity of prefetching can undermine the benefits of prefetching itself. In multi-core processor, either helper threads or another core pulls data into a cache (typically to L2 cache) that is shared by a core that is running prefetching algorithm with processing cores. There has been no special hardware support for helper threads or another core to observe the patterns of processing core. To compensate this, fragments of code has been passed or compiled on helper threads, which has the synchronization problem as mentioned earlier. In DPS, we support to use minor modifications to the microarchitecture of memory subsystem to support observing data access patterns of processing cores. We also introduced an addition of prefetch cache, where predicted data is directly pushed. The CPU accesses this prefetch cache simultaneously with L1 cache. Prefetch cache hits are a lot faster than shared L2 cache prefetch hits of *pull-based* strategies.

Another benefit of our DPS is prefetching engine's choosing prediction algorithms based on history. This gives adaptability to select prediction algorithms with various complexities according to running application's data access pattern, instead of sticking to a single algorithm. As there is no universal algorithm that can predict all access patterns with the same complexity, our approach is beneficial. As shown in Table 3.1, various algorithms can be used by DPS. It is also flexible to use any new algorithms that may be devised in the future, since DPS system is feasible to implement as a thread.

DPS can support multiple processing cores. Two proposed research projects [Solt02, Hafe04] use memory processor approach to push data. Solihin et al. use a memory processor to push data into L2 cache, while Hassanein et al. push data into L1 cache or

registers. However, both are plagued by coupling the prefetcher to memory. The first strategy observes L2 cache misses at a processor inside memory and pushes predicted data into L2 cache. The latter approach uses pre-execution of code in memory processor to predict future accesses. However, these two methods support only one core. As the number of cores to be supported increase, the workload on memory processor increases and it is highly probable that the performance gains degrade. Our DPS approach, however, uses cores that are dedicated for running prefetching algorithms. One might argue that running computing on both processors in a dual-core processor is more beneficial than one core supporting data access prefetching of another. However, our DPS is designed to support prefetching in larger processors, with more cores. We used dual core processor to explain the design of DPS. Benefits of DPS are useful for future HEC processors, where more processing cores require data access service. It is commonplace in HEC computing to have idle nodes. Similarly, in future HEC processors, we could designate these idle cores to execute the operation of Data Push Server. As DPS is a software system with hardware support, processing cores have a choice not to use DPS's service, if there are no idle cores.

3.8 CASE STUDY: IMPLEMENTING DPS ON CELL PROCESSOR

Implementation of DPS can be done with a combination of software with hardware support. Many current processors support multiple threads to run on each processor core. DPS can be implemented on a helper thread that has interface to monitor data access history buffer and to push data into prefetch cache. The goal of DPS is to serve multiple processing cores that are clients for DPS service as shown in Figure 3.6. The thread level

implementation with hardware support provides that flexibility to serve multiple cores of a processor by instantiating separate thread for each client. Emerging chip-level multiprocessors (IBM's Cell processor [Ibmc06], ClearSpeed's co-processors [Clea06] etc.), have many processing cores. It is not far in the future we will witness many general purpose processors containing hundreds of cores in each processor. Using some of these cores specifically to run DPS system serving other processing cores is beneficial.

In the previous sections, we discussed the modifications to be done to processor microarchitecture and to memory subsystem in multicore processors. Among existing processors, Cell processor [Ibmc06] has a positive environment for implementing our DPS without hardware modifications. As shown in Figure 3.14 Cell processor microarchitecture is a heterogeneous multi-core architecture, which has one PowerPC Processing Element (PPE) and eight Synergistic Processing elements (SPE). Each SPE has a software-controllable local store (LS) of 256 KB size. SPEs and PPE share an external system memory. Each SPE has a synergistic memory flow controller (MFC), which connects SPEs to each other and to PPE, through a very high bandwidth Element

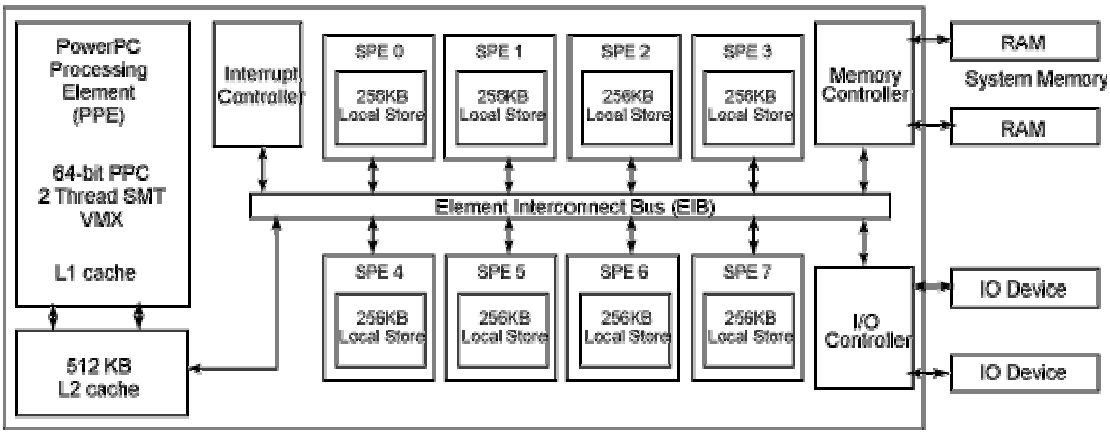


Figure 3.14. IBM Cell Broadband Engine Architecture

Interconnect Bus (EIB). MFCs perform data movement among individual SPEs and PPE. They provide the primary method for transfer, synchronization, and protection between main memory and local storage. MFC commands provide an interface to perform the data transfers. These are referred to as MFC DMA commands. Each MFC can support multiple DMA transfers at the same time by maintaining queues of MFC commands.

The provision of SPEs is a great benefit for computing intensive SIMD parallel applications, which are instruction-intensive workloads. When it comes to data intensive applications, as many HPC applications, the performance of Cell processor as many multicore processors is limited by data latency rather than instruction latency. Prefetching data into local stores of SPEs can improve the scope of Cell processors to provide superior performance improvement for data intensive applications. PPE has a Data Prefetch Engine (DPFE) to prefetch data into PPE's cache memory. The DPFE performs simple sequential and regular strided predictions for cache memory of PPE. Prefetching for SPEs can be done by software, where either program developers or compilers provide hints for prefetching. This software controllability of SPE local store provides a scope for improving I/O performance of parallel applications as well with the use of DPS.

Challenging issues of implementing DPS system on any processor are the interface to collect data access history and to push data closer to the processor. We can use software implementation to predict future accesses, which is practicable once we have prediction strategy algorithms such as strided pattern prediction by using multi-level difference table introduced in sections 3.4 and 3.5.

On Cell processor, the prediction algorithms of DPS system can be executed on PPE. The collection of data access patterns and pushing data into SPEs can be performed by

the use of software libraries. IBM provides various Cell processor SPE C/C++ language extensions [Cbeh06]. These libraries provide interface to access SPE local store directly from a program running on PPE. The collection of data access patterns can be done by running a daemon on each SPE to that sends the list to the PPE over either over DMA or by using effective-address aliases to a local store (LS) with privileged software on PPE (section 3.1, page 64 of BE Handbook, v.1.0). Over DMA, we can use DMA list commands provided by SPE C/C++ extensions [Cbeh06].

To predict future data references, DPS chooses prediction algorithm adaptively based on collected data access history. As shown in Figure 3.6, PPE acts as DPS core. To push data after predicting, PPE initiates a DMA request to put data into LS of SPE. Current language extensions from IBM provide interface to put data into LS using *put_block ( )* function. To put multiple blocks at a time, this function can be extended, called *put_blocks ( ),* which is similar to existing Memory Flow Control DMA commands and MFC DMA list commands. If DMA latency is too high, Cell processor provides another solution, where PPE can have direct access to store predicted data into LS of an SPE directly, using effective address aliases (Section 3.1, page 64 of BE Handbook, v.1.0) [Cbeh06]. DPS system can run as privileged software to access the LS of SPEs to push predicted data blocks into LS. With simple software development, DPS can be executed on Cell processor to improve the data access performance of data intensive HEC applications for future Cell based supercomputers.

3.9 SUMMARY

In this chapter, we presented the design of our server-based data push architecture. We discussed various features and functionality of DPS, whose goal is to proactively push data closer to the CPU based on predicting future references. We first discussed the issue of monitoring data and selecting prediction algorithm based on a comprehensive classification of data access patterns. We present the design of a data access history buffer to hold history of data accesses at L1 cache and an instruction called GDAH to collect it into a DPS system. DPS then predicts future references as well as their access time to provide *in time* prefetching. Data Propeller module of DPS pushes data into a separate prefetch cache. To maintain coherence, we use a validation component that discards prefetched cache lines if those lines are already available in caches.

DPS has many benefits by separating data access from computing. These include isolation of complexity from computing cores, performance gains, and supporting multiple cores with a scalable DPS. We discussed the implementation of DPS using modifications to microarchitecture of memory subsystem. We described the modified CPU data access operation. We also studied the possibility of implementing DPS on IBM Cell processor, which provides software controllable local store in place of L1 cache. On the Cell processor, DPS can be implemented on PowerPC Processing Element (PPE), without any hardware modifications.

In the next chapter, we discuss modeling the data access cost and optimizations at software level. In Chapter 6, we present the simulation of DPS on SimpleScalar simulator and the performance results for various SPEC CPU2000 benchmarks that have poor cache performance.

CHAPTER 4

MODELING DATA ACCESS PERFORMANCE

In the previous chapter, we introduced the Data Push Server framework to provide accurate and in-time prefetching with aggressive spatial and temporal prediction strategies. We also presented operation of memory subsystem in multicore processor. In this chapter, we discuss data access optimization methods at software level, and models to predict data access performance. We then present Simple Memory Access Cost (SMAC) prediction model, which is based on various data access patterns and the size of data accesses to help applying cache reuse optimizations in high performance applications.

4.1 MEMORY PERFORMANCE MODELS

Cache memories are characterized by their size, line size and associativity. Cache size (C) represents its capacity in bytes. Caches are organized in cache lines. The data transfer between two levels of memory hierarchy is done based on the size of these lines. When a cache miss occurs, a block of data of size equal to cache line size (L) is fetched from the next lower level cache or memory. This property conforms to spatial locality. Associativity of a cache helps in deciding how many locations are there to place a cache line. For a *direct mapped* cache (1-set associativity), a fetched cache line can be placed in one of the (C / L) positions. If another cache line is mapped to the same location, currently residing cache line has to be replaced. Higher associative caches provide more than one place to map a cache block. In a *fully associative* cache, a cache block can be placed anywhere in the cache. But *fully associative* caches are costly. It is common for

architectures to have 2-way to-8-way associative caches. In an 8-way associative cache, a block can be placed in any of the 8 positions in each set of 8 lines.

Many computation models for parallel program performance exist and they provide simple classification of communication performance [Mamt95]. The Hierarchical Memory Model (HMM) applies the characteristics of memory hierarchies to network communication. The cost estimates of this model are accurate for very large sets of streaming data, but ignore the network attributes that are common in parallel processing. Other models such as Bulk Synchronous Parallel (BSP) [Vali90] and LogP [Ckps96] models quantify communication latency and bandwidth. LogP is widely used in parallel computing as it also incorporates the asynchronous behavior and communication overhead. Many extensions of LogP model are available [Mofr98, Aiss95]. However, all these models ignored the effect of memory latency and the middleware latency that might occur before network communication. This latency is caused due to the data collection overhead when the data has to be gathered from user memory. This overhead was small compared to the network latency historically when the networks were slow and was considered as a constant. But with the advances in network technology, the memory overhead has become significant. It is also not wise to consider this overhead a constant as it heavily depends on the data access pattern and size.

Towards the goal of modeling this overhead, we first classified the communication cost as *memory communication* and *network communication*. Memory communication (or memory copying) is the transfer of data from the user's buffer to the local network buffer (or shared-memory buffer) and vice versa. Network communication is the movement of data between source and destination network buffers.

The memory-communication cost for sending a data segment depends on architectural parameters, such as cache capacity, and code characteristics, such as data distribution. In general, the overall communication cost includes data-collection overhead, the cost of data copying to the network buffer, the cost of data forwarding to the receiver (network-communication cost), and other costs added by the middleware implementation. When data distribution in memory is noncontiguous, the data is typically collected into a contiguous buffer before being copied to the network buffer. This process adds extra buffering overhead to the overall communication cost and is implementation dependent.

We can quantify the memory-communication overhead as follows. We divide the overall communication cost into three parts: basic contiguous data-copying cost (o), memory-communication cost (l), and the network-communication cost (L). The memory-communication cost is further classified as data-packing overhead (l_p) and middleware-induced overhead (l_m). The middleware-induced overhead includes all other costs, such as extra buffer-copying cost, handshaking overhead between source and destination processes (if there is any), and load-imbalance costs.

$$Communication\_overhead = o + l_p + l_m + L$$

We measured each of these costs as follows. The basic data-copying cost (o) is the cost of copying a contiguous data between two buffers. The network-communication cost (L) is calculated by subtracting (o) from the cost of communicating a contiguous message between two processes. The cost of reading data noncontiguously from a buffer and writing it into a contiguous buffer includes data-packing cost (l_p) and the basic overhead (o). Subtracting (o) from this cost gives the data-packing cost (l_p). The middleware-

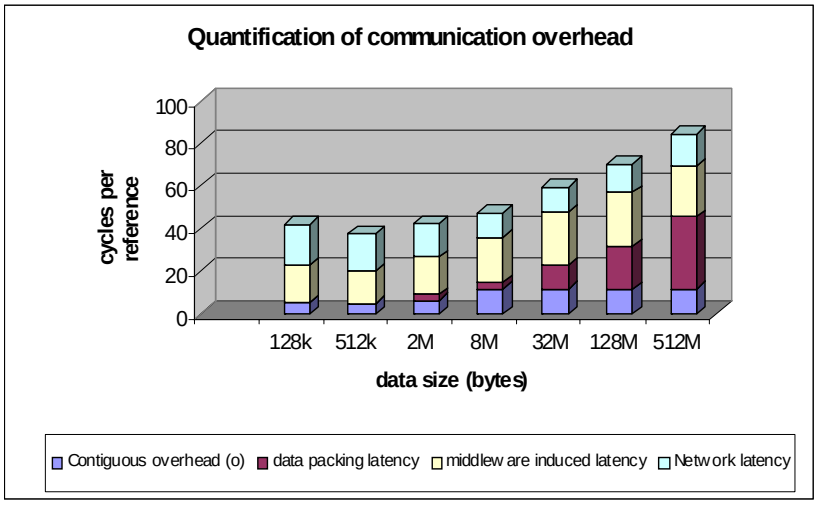


Figure 4.1. Memory-communication cost for a matrix-transpose algorithm is classified into basic contiguous overhead, data-packing cost, and middleware-induced cost. The network-communication cost is also shown.

induced cost (l_m) is calculated by subtracting the sum of other costs from the overall communication cost. The results for a parallel matrix-transpose algorithm with derived datatypes are shown in Figure 4.1. It shows that the basic-copying cost (o) is relatively constant per data reference. In contrast, data-packing and middleware-induced costs grow significantly with data size. In this chapter, we show how to reduce the data-packing cost of the overall memory-communication cost.

This quantification led to new models of parallel communication to extend LogP model by Cameron and Sun, called memory-logP model [Casu03]. The memory-LogP model formally characterizes the memory-communication cost under four parameters: l: the effective latency, defined as the length of time the processor is engaged in transmission or reception of a message due to the influence of data size (s) and distribution (d), l=f(s, d); o: the overhead, defined as the length of time the processor is engaged in transmission or reception of an ideally distributed (contiguous) message

(during this time, the processor cannot perform other operations); g: the gap, defined as the minimum time interval between consecutive message receptions at the processor (the reciprocal of *g* corresponds to the available per processor bandwidth for a given implementation of data transfer on a given system); and P: the number of processor/memory modules (point-to-point communication in the memory hierarchy implies P=1). This model was further extended by Cameron to include middleware overhead [Cage04].

With this understanding, we extend our research to find the memory access cost based on prediction instead of classifying the communication cost repeatedly. Our model aims to predict the memory access cost for various data access patterns that occur in numerical and scientific applications. In the following section, we first explain the data access patterns and then the prediction model.

4.2 DATA ACCESS PATTERNS

In chapter 3, we gave a classification of data access patterns based on strides between accesses. In this section, we explain more on our classification using strides and size of data blocks that are being accessed. Loops and arrays are fundamental structures of most numerical and scientific applications [Peak98]. A major share of the execution time of these applications is spent in loops, accessing data from arrays. Loop variables are incremented or decremented to find the reference of an array. Analyzing these reference patterns of array accesses is needed to find out the hotspots and to optimize the performance by reorganizing these memory references.

Data access patterns are classified based on the stride between successive accesses. Modal model of memory [Mitc01] categorize data accesses as constant, strided and non-

monotonic modes. Yan et al. [Ryan02] classify memory access patterns into three types: migratory, group and unpredictable patterns.

We classify data access patterns in scientific applications as constant, contiguous and non-contiguous. Non-contiguous pattern is further divided into four patterns. This classification is based on the size of data blocks accessed with each reference and their successive strides. Stride is the distance between the previous reference and current reference.

Constant accesses are those where the same data block is accessed repeatedly i.e., stride is equal to zero. In this type of accesses, once the data block is loaded into the cache, further accesses would not cause any other cache misses.

Contiguous access pattern is where the stride between successive accesses is equal to the size of datatype. These are divided further as fixed length block accesses and variable length block accesses. Fixed length block accesses refer to the same datatype in consecutive references.

Non-contiguous access pattern is where the stride of next reference is greater than the size of currently accessed datatype. These can be further divided as follows:

a. *Fixed length block, with fixed stride*: Stride is similar through out the access pattern. As shown Figure 4.2.a., a block with a size of constant block_size is copied into dest from src. The next block is copied from src+stride to *dest+block_size*, i.e. array src is being accessed non-contiguously with a fixed stride and array dest is being accessed contiguously.
b. *Fixed length block, with varying stride*: The stride varies between each access. (Figure 4.2.b.)

c. *Variable size block, with fixed stride*: Accessing different or varying size datatypes, where the strides of accesses are similar. (Figure 4.2.c.)

d. *Variable size block, with variable stride*: Accessing different or varying size datatypes, where the strides of accesses are varying. (Figure 4.2.d.)

These access patterns cover all the access patterns of data accesses in scientific applications.

```
for (i=0; i < n; i += stride)
{
        memcpy(dest, src+i, block_size);
        dest += block_size;
}
```

Figure. 4.2.a

```
for (i=0, j=0; i < n; i += stride[j])
{
        memcpy(dest, src+i, block_size) ;
        j++ ;
        dest += block_size;
}
```

Figure. 4.2.b

```
for (i=0, j=0; i < n; i += stride)
{
        memcpy(dest, src+i, block_size[j]) ;
        dest += block_size[j];
        j++ ;
}
```

Figure. 4.2.c

```
for (i=0, j=0; i < n; i += stride[j])
{
        memcpy(dest, src+i, block_size[j]) ;
        dest += block_size;
        j++ ;
}
```

Figure. 4.2.d

Figure 4.2. Noncontiguous Data Access Patterns

4.3 SIMIPLE MEMORY ACCESS COST (SMAC) PREDICTION MODEL

Model parameters:

As we explained before, a cache memory is characterized by its size, cache line size and associativity. In our model, we consider TLB as a level of memory hierarchy. Its parameters are page size P (similar to cache line size of a cache) and the capacity. The capacity of a TLB is the amount of memory page mapping it can store and is equal to number of page entries multiplied by page size.

Table 4.1 summarizes the memory hierarchy parameters. Subscript i of a parameter signifies the level of that cache/TLB in the hierarchy of memory. Cache memory at level i has three properties: its size in bytes (C_i), cache line size (L_i) and its associativity (A_i). TLB is represented with the number of page table entries (T_s), page size (P_s) and it associativity (A_T). M refers the total number of cache levels.

Table 4.1. Memory hierarchy parameters

C_k	Cache size at k^{th} level cache of memory hierarchy
L_k	Cache line size at k^{th} level cache of memory hierarchy
A_k	Associativity of k^{th} level cache of memory hierarchy
M_k	Number of cache misses at k^{th} level cache of memory hierarchy
$M^{c}_{(k,i)}$	Number of cache misses at k^{th} level cache of memory hierarchy in accessing i^{th} array, contiguously.
$M^{n}_{(k,i)}$	Number of cache misses at k^{th} level cache of memory hierarchy in accessing i^{th} variable, non-contiguously.
T_s	Number of page table entries (PTE) in TLB
P_s	Page size of each PTE
A_T	TLB associativity
M	Number of cache levels in memory hierarchy

Cache misses are classified into three types [Mark87]. *Compulsory misses* are the misses that occur when the CPU accesses a data block for the first time. *Capacity misses* are the misses that occur due to insufficient cache to hold an entire range of data that is being accessed at a time. This range is also called working-set. *Conflict misses* are the misses that occur when more than one cache block maps to the same location even though the existing cache block still is being used in the near future. Cache misses at level *i* are represented with M_i. $M^{c}_{(k,i)}$ refers to the number of cache misses at level *k* of memory hierarchy, in accessing ith array (variable) contiguously. If it is being accessed non-contiguously, it is represented by $M^{n}_{(k,i)}$.

Table 4.2. Data access parameters

$R^{c}_{(k,i)}$	Number of contiguous references of i^{th} array at cache level *k* of the memory hierarchy.
$R^{n}_{(k,i)}$	Number of non-contiguous references of i^{th} array at cache level *k* of the memory hierarchy.
W_i	Fixed size of the data block being accessed in i^{th} array.
S_i	Fixed stride of accessing i^{th} array non-contiguously.
W_i^c	Variable size of the data block being contiguously accessed in i^{th} array.
$W^{n}_{(i,j)}$	Variable size of j^{th} data block being non-contiguously accessed in i^{th} array.
$S_{(i,j)}$	Variable stride of the j^{th} data block being contiguously accessed in i^{th} array. ($stride_j$ in stride signature)
D	Size of working set

Data access pattern parameters are shown in Table 4.2. The subscript i represents the ith array being accessed. The parameters $R^{c}_{(k,i)}$ and $R^{n}_{(k,i)}$ represent the number of contiguous and non-contiguous references separately. S_i is the fixed stride in accessing the ith array and $S_{(i,j)}$ is the variable stride. D is the working set size and W_i represents the block (word) size of the ith array.

Our goal is to predict the memory access cost of a basic block of loop with any type of data access patterns discussed in section 2, and for multiple data array variables. We assume LRU replacement policy for cache and TLB. We assume that the memory hierarchy is following inclusive property. The total cost of accessing memory includes the access time and the miss penalties of these levels in the hierarchy. If there are k levels of cache memory and one level TLB [Patt96],

*Total Memory cost = (Number of TLB hits * Time to access TLB) +*

*(Number of TLB misses * TLB miss penalty) +*

*(Number of L1 hits) * (Time to access L1) +*

*(L1 misses * L1 penalty) + (L2 misses * L2 penalty) + … +*

*(Lk misses * Lk penalty)* … (4.1)

To predict this cost, we have to find the number of cache hits/misses at each level and TLB hit rate. We predict the cache and TLB misses based on the access pattern.

Assuming that there are M levels of cache, the total miss penalty due to cache misses is the sum of miss penalty at each level.

$$T_m = \sum_{k=1}^{M} (M_k * T_k) - \alpha \quad \ldots \quad (4.2)$$

where M_k is the total number of cache misses and T_k is the miss penalty at level k cache. α is the overlapping the cache misses with prefetching and other OS optimizations.

Consider that there are m array variables accessed contiguously and n array variables accessed non-contiguously, the total number of misses at cache level k is the sum of misses caused in accessing contiguously accessed arrays and those of non-contiguously accessed arrays.

$$M_k = \sum_{i=1}^{m} M^{c}_{(k,i)} + \sum_{i=1}^{n} M^{n}_{(k,i)} \quad \ldots \quad (4.3)$$

$M^{c}_{(k,i)}$ is the number of cache misses at k^{th} level cache of memory hierarchy in accessing i^{th} variable, contiguously. $M^{n}_{(k,i)}$ is the number of cache misses at k^{th} level cache of memory hierarchy in accessing i^{th} variable, non-contiguously.

Now we count the number of cache misses based on the data access pattern.

Constant access pattern: In this type of accesses, once a word is loaded into the cache, the following accesses to the same word cause no extra cache misses. If the word size of a variable is W_i and there are the number of cache misses is equal to $\lceil (W_i / L_k) \rceil$.

Contiguous access pattern: In this pattern, the stride between successive accesses is the same as data size. All the cache misses caused in this pattern are compulsory misses. Each reference fetches a cache line into the cache. Cache line contains more than one word of data. This is to assure spatial locality property of using cache memory. If the cache line size is more than the data type accessed, the next reference utilizes the prefetched data from the cache. Each reference causes $\lceil (W_i / L_k) \rceil$ misses, i.e. if the word size is more that cache line size, then it causes more than one miss, otherwise just one miss occurs for every $\lceil (L_k / W_i) \rceil$ references. If there are *n* references, the number of cache misses caused at cache level k in accessing variable i is:

$$M^c_{(k,i)} = \lceil n * (W_i / L_k) \rceil$$

If i^{th} variable has $R^c_{(k,i)}$ references, the number of cache misses is:

$$M^c_{(k,i)} = \left\lceil R^c_{(k,i)} * (W_i^c / L_k) \right\rceil \quad \ldots \quad (4.4)$$

where W_i^c is size of the data block being contiguously accessed in i^{th} variable.

The number of cache references at level k ($R^c_{(k,i)}$) is the number of cache misses at the lower level cache, i.e. $R^c_{(k,i)} = M^c_{(k-1,j)}$.

Non-contiguous access patterns: As described in previous section, there are four main types of access patterns. These patterns are classified based on the variability of stride and data block size. The occurrence of cache misses is categorized into four regions based on the working set size, similar to Saavedra and Smith [Saav95]. First region is the one where all the working set fits in the cache. As long as the working set size is less than the cache size, the total data fits into the cache. All the cache misses are

compulsory misses. This number is equal to $M^{n}_{(k,i)} = \lceil n*(W_i / L_k) \rceil$ at level k of memory hierarchy in accessing i^{th} variable non-contiguously. This number is the same for all types of non-contiguous access patterns.

When size of the data working set exceeds the cache size, three regions of memory operations are defined. The first region is when the stride (S) is between 1 and cache line size ($1 < S \le L_k$). The second region is $L_k < S \le D / A_k$, where A_k is the associativity of k^{th} level cache of memory hierarchy. The third region is $D / A_k < S \le D / 2$. In this last case, although the $D > C_k$, only $D / S < A_k$ amount of data is needed for access. In the last region the number of references mapping to a single set is less than the set associativity. Thus, only compulsory misses are caused in the third access pattern, i.e.

$$M^{n}_{(k,i)} = \lceil n*(W_i / L_k) \rceil) \qquad \ldots \qquad (4.5)$$

where n is the number of data accesses. Thus, we set our focus on the first two regions to count the number of cache misses.

First we find the cache misses for a fixed size of data block accesses of one variable, with a fixed stride.

If the stride (fixed) is less than the cache line size, one cache miss occurs for (L_k / S) references. If there are n references, the number of cache misses is: $n*(S / L_k)$, where n is the number of data accesses.

If the stride (fixed) is more than the cache line size, each reference causes $(\max(\lceil W_i / L_k \rceil, 1)$ cache misses, i.e. each access causes one miss when the word size is less than L_k . If word size is more than L_k , each reference causes $\lceil W_i / L_k \rceil$ misses. If there are n references, the number of cache misses is equal to $n*(\max(\lceil W_i / L_k \rceil, 1)$.

$$M^{n}_{(k,i)} = R^{n}_{(k,i)} * (\max(\lceil W_i / L_k \rceil, 1) \quad \ldots \quad (4.6)$$

For variable stride with fixed size block accesses, the cache misses have to be counted for each stride. If the stride is less than L_k, it does not cause a cache miss as the pre-fetched line of data is reused. The number of cache misses is:

$$M^{n}_{(k,i)} = (\sum_{j=1}^{R^{n}_{(k,i)}} \lfloor \min((S_{(i,j)} / L_k), 0) \rfloor) * (\max(\lceil W^{n}_{(i,j)} / L_k \rceil, 1)) \quad \ldots \quad (4.7)$$

$M^{c}_{(k,i)}$ is the number of cache misses at k^{th} level cache of memory hierarchy in accessing i^{th} variable, non-contiguously, $S_{(i,j)}$ is variable stride of the data block being contiguously accessed in i^{th} variable. $R^{n}_{(k,i)}$ is the number of non-contiguous references of ith array at cache level k of the memory hierarchy. $W^{n}_{(i,j)}$ is the size of j^{th} data block being non-contiguously accessed in i^{th} variable. In this pattern when the stride $S_{(i,j)}$ is less than the cache line size, we assume that the cache line has already been fetched into the cache. However, when this stride is causing to fetch a new cache line, then this formula misses to count that cache miss. This can be corrected by maintaining the history of cache line that has been fetched recently.

The number of cache references at level k ($R^{n}_{(k,i)}$) is the number of cache misses at the lower level cache, i.e. $R^{n}_{(k,i)} = M^{n}_{(k-1,j)}$.

For fixed or variable stride with variable size block accesses, the cache misses have to be counted for each block size. In this case, we assume that the stride is always more than L_k. If the data block size is less than L_k, it does not cause a cache miss as the pre-fetched line of data is reused. The number of cache misses is:

$$M^{n}_{(k,i)} = \sum_{j=1}^{R^{n}_{(k,i)}} (\max(\lceil W^{n}_{(i,j)} / L_k \rceil, 1) \quad \ldots \quad (4.8)$$

Refer Table 4.3 for a summary of formulae to calculate the cache misses for all data access patterns. Using (4.3) total number of cache misses in accessing contiguous and non-contiguous data is calculated. Formula 4.2 gives the total memory access cost.

Table 4.3. Number of cache misses for all data access patterns

Data access pattern		Number of cache misses
Constant		$M^{c}_{(k,i)} = \left\lceil (W^{c}_{i} / L_k) \right\rceil$
Contiguous		$M^{c}_{(k,i)} = \left\lceil R^{c}_{(k,i)} * (W^{c}_{i} / L_k) \right\rceil$
Non-contiguous ($D < C_k$)		$M^{n}_{(k,i)} = \left\lceil R^{n}_{(k,i)} * (W^{n}_{i} / L_k) \right\rceil$
Non-contiguous ($D > C_k$)	$1 < S \le L_k$	$M^{n}_{(k,i)} = \left\lceil R^{n}_{(k,i)} * (S / L_k) \right\rceil$
	$L_k < S \le D / A_k$	$M^{n}_{(k,i)} = R^{n}_{(k,i)} * (\max(\lceil W_i^{n} / L_k \rceil, 1)$
	Variable stride, fixed data block size	$M^{n}_{(k,i)} = (\sum_{j=1}^{R^{n}_{(k,i)}} \lfloor \min((S_{(i,j)} / L_k), 0) \rfloor) * (\max(\lceil W^{n}_{(i,j)} / L_k \rceil, 1))$
	Variable stride ($L_k < S \le D / A_k$), variable data block size	$M^{n}_{(k,i)} = \sum_{j=1}^{R^{n}_{(k,i)}} (\max(\lceil W^{n}_{(i,j)} / L_k \rceil, 1)$
	$D / A_k < S < D / 2$	$M^{n}_{(k,i)} = \left\lceil R^{n}_{(k,i)} * (W^{n}_{i} / L_k) \right\rceil$

4.4 MODEL VERIFICATION

This section presents performance measurements to verify the predicted memory access cost with the measured cost on various architectures. We measure the performance of loops with all the data access patterns mentioned above and compare that performance with the predicted performance.

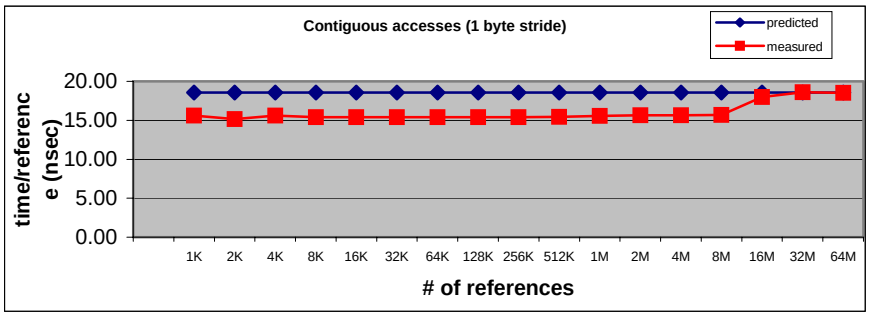
We took the measurements on a Sun Solaris based cluster called Sunwulf, which is located at the Scalable Computer Software Lab of Illinois Institute of Technology. *Sunwulf* is composed of a four-processor E450 server and 63 high-end workstations. We run our experiments on one of the nodes. Each node is a SUN Blade-100 workstation with one UltraSparc-IIe, 500MHz CPU. The L1 cache is 16KB, with a 16-byte cache line size. The L2 cache has a capacity of 8MB and its line size is 64 bytes. It also has a TLB with 4KB page size and 48 entries. We used a microbenchmark to find the average access time and miss penalty of each level of memory hierarchy. This is similar to the microbenchmark proposed by Saavedra and Smith [Saav95].

Another platform we used for experiments is a 32-node Beowulf, located at University of South Carolina. Each node consists of 933MHz, Pentium III processor. It has 16 KB L1 cache and 256KB L2 cache. Both these caches are on the die, and the average penalty for load misses is measured as 7 cycles and 70 cycles for L1 and L2 respectively. We chose these processors, as they apply inclusive property in the memory hierarchy with less aggressive pre-fetching.

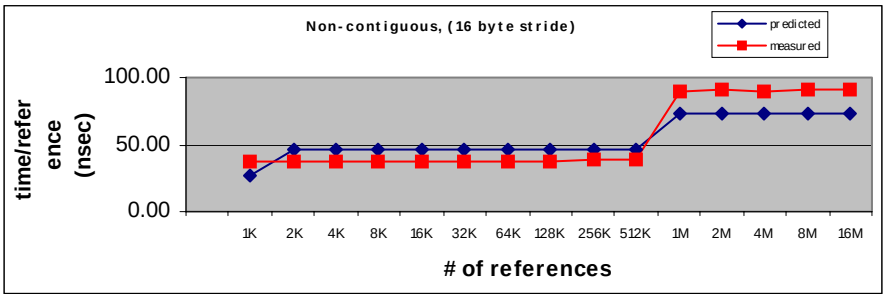
We used the loops similar to Figure 4.2 and measured the time to execute those loops. In all these loops, two array variables are accessed with different access patterns. We chose these loops since many applications contain loop blocks where the data accesses are similar to the access patterns discussed above. We can apply the same prediction model for any number of arrays. Execution time of these loops contains only the data access cost, without any computation cost. We used pointer-to-pointer copy to avoid the cost of memcpy. In these experiments, we ran many iterations of the program to find the minimum cost. We also flushed the cache after measuring the time for an iteration to

replace any cache blocks that are reusable. We compiled these programs using gcc 3.0 and padded the arrays to avoid any cache thrashing. The comparison of predicted cost and measured memory access cost is presented in the following paragraphs. The memory access cost is presented as a ratio of execution time to the number of memory references. This normalization is done to fit all the data into the graph. The performance is better for lower values.

Figures 4.3 and Figure 4.4 compare the predicted memory access cost with measured cost in running the loops in various data access patterns explained in section 4.2 (Figure 4.2) on Sunwulf cluster. For contiguous data accesses (Figure 4.3.a.), the predicted cost is constant per reference. The prediction error reduced as the number of references



4.3.a.

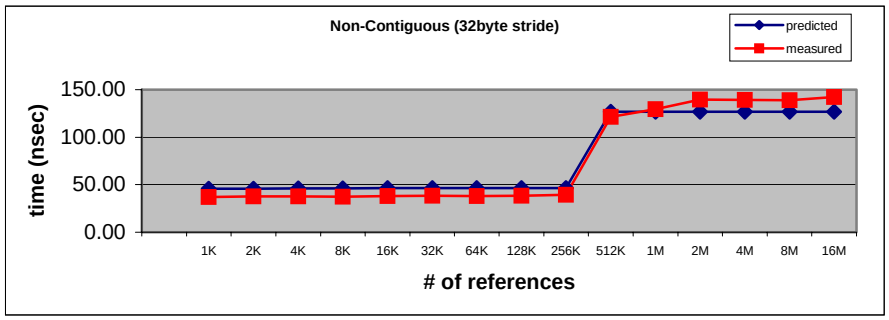


4.3.b.

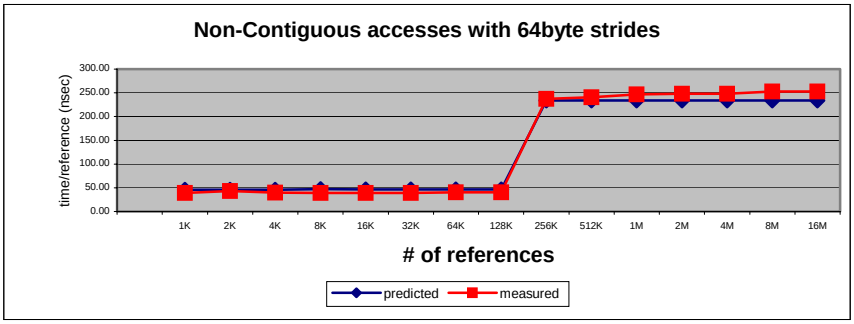
Figure 4.3. Comparison of measured and predicted memory access cost. The access patterns are: 4.3.a. Contiguous data access (word size: 1byte, stride: 1byte). 4.3.b. Non-contiguous data access with fixed word size and stride (word size: 8 bytes, stride: 16 bytes)

increased. The error was mainly due to the approach of counting cache misses pessimistically without taking prefetching into consideration. The prediction error was below 20% for small data and below 4% for large data with this data access pattern.

To test the non-contiguous access pattern performance we used three sizes of fixed strides (16bytes, 32 bytes and 64 bytes) that are equal to L1 cache line size, more than L1 line size and that of equal to L2 line size. For non-contiguous accesses, with stride equal to L1 cache line size, the prediction error reduced as the data size increase. It can be seen from Figure 4.3.b and Figure 4.3.c, that the utilization of caches are more effective when the data size is less than L2 cache size. Overall, the error is below 20% in most of the



4.3.c.



4.3.d.

Figure 4.3. Comparison of measured and predicted memory access cost. The access patterns are: 4.3.c. Non-contiguous data access with fixed word size and stride (word size: 8 bytes, stride: 32 bytes) 4.3.d. Non-contiguous data access with fixed word size and stride (word size: 8 bytes, stride: 64 bytes)

cases. For the remaining two non-contiguous access patterns with fixed strides, the prediction error is below 10% for larger data sizes.

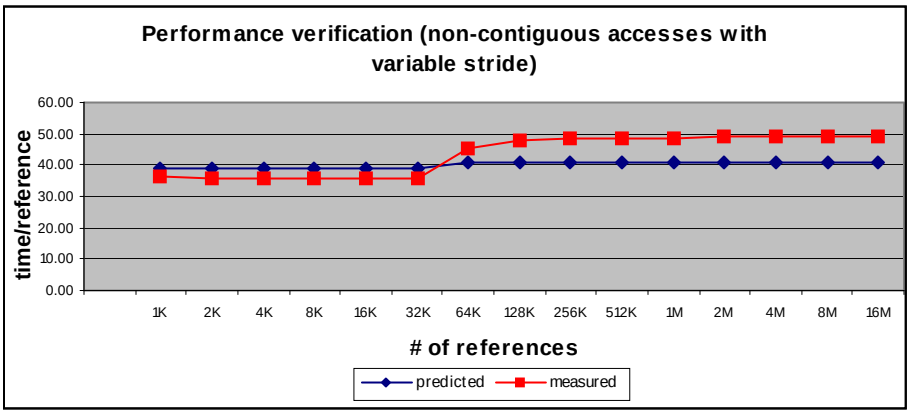
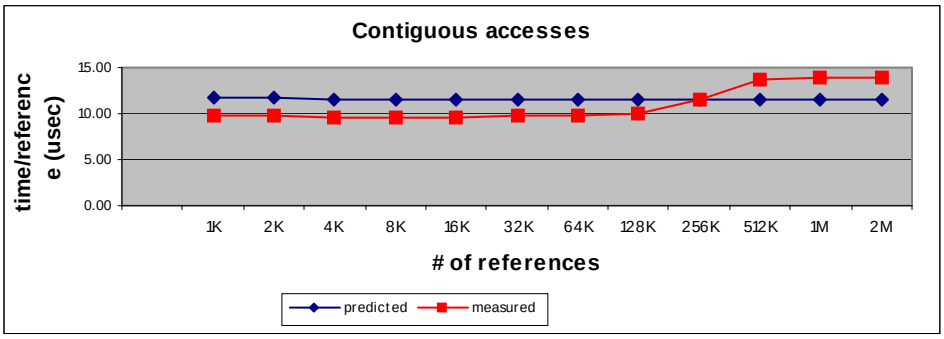


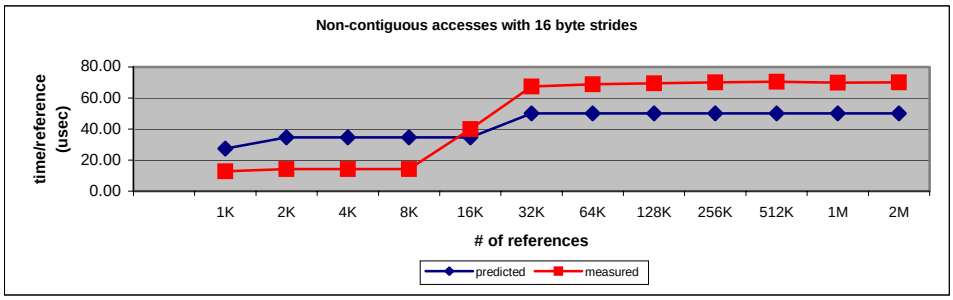
Figure 4.4. Comparison of measured and predicted memory access cost for non-contiguous data access with fixed word size and variable strides (word size: 8 bytes, stride varies from 1 to 128 bytes periodically)

For non-contiguous access pattern with variable strides, we initialized an array that contains strides of accesses. Prediction cost of this access pattern contains the cost of accessing non-contiguous arrays as well as the cost of accessing the array of strides. The prediction error is below 15% (Figure 4.4). This error is caused by missing some of the cache misses in non-contiguous accesses, which requires maintaining the history of the length of cache lines that are already been fetched into the cache. Another reason for prediction error for all these access patterns is that we are using average miss penalties, which may not be accurate.

We observe the similar results on Pentium III processor on Beowulf cluster (Figures 4.5, 4.6, and 4.7). The prediction error is slightly high for small data sizes where the prefetching of this processor is effective. As the working set size increase, the L2 misses increase and the prediction error is below 20% in these cases.



4.5.a.

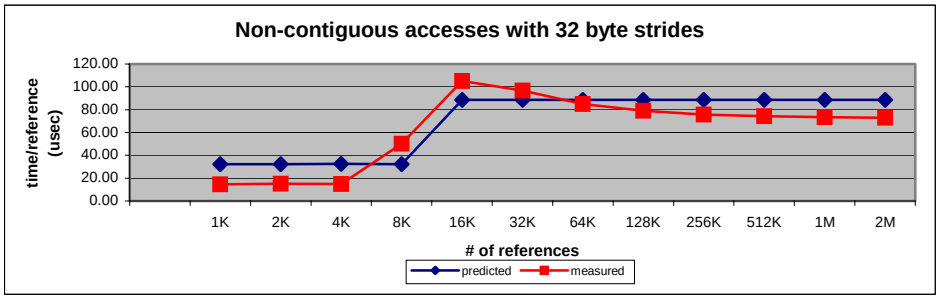


4.5.b.

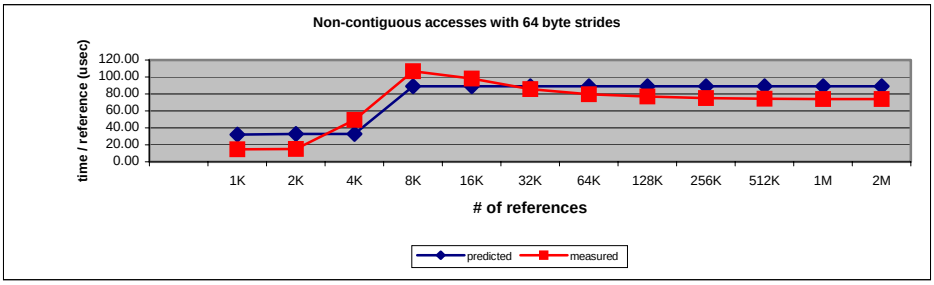
Figure 4.5. Comparison of measured and predicted memory access cost on Pentium III processor. The access patterns are: 4.5.a. Contiguous data access (word size: 1byte, stride: 1byte). 4.5.b. Non-contiguous data access with fixed word size and stride (word size: 8 bytes, stride: 16 bytes),

We also verified the performance of the loops in NAS Parallel benchmarks that are performing matrix transpose operation. We have measured the performance two variations of matrix transpose algorithms from NAS Parallel benchmarks' Fast Fourier Transform program. The first algorithm is a simple matrix transpose of copying rows of one matrix to columns of another matrix. The second algorithm uses cache-blocking optimization to improve the performance. Both algorithms fit into the data access patterns explained in section 4.2. The data working set of the first algorithm increases with the dimension of the matrices. Due to the row major ordering of arrays in C (column major ordering in Fortran), one matrix is accessed contiguously and the other is accessed non-contiguously with fixed stride. The second algorithm makes sure that a block of data is

fully utilized before replacing it from the cache. In this algorithm, the two matrices are accessed non-contiguously with fixed strides. However, as the whole data block is reused before it is being replaced, and we chose the block size such that it fits into the cache, the number of cache misses is very less compared to the unoptimized version of matrix transpose. These experiments are performed on Sun UltraSparc IIe processor node.



4.6.a.



4.6.b.

Figure 4.6. Comparison of measured and predicted memory access cost on Pentium III processor. The access patterns are: 4.6.a. Non-contiguous data access with fixed word size and stride (word size: 8 bytes, stride: 32 bytes) 4.6.b. Non-contiguous data access with fixed word size and stride (word size: 8 bytes, stride: 64 bytes)

As expected, the performance (time/reference) increases as the data size increases for the unoptimized transpose algorithm (Figure 4.8). Predicted values of performance are slightly different from the measured values. The error is around 13%. In the second

algorithm, the performance is improved for the transpose algorithm due to the cache-blocking optimization (Figure 4.9). The performance error was below 5% for most of the data sizes, but increased for large data sizes. This is mainly due the increase in average time per memory reference for the large data sizes.

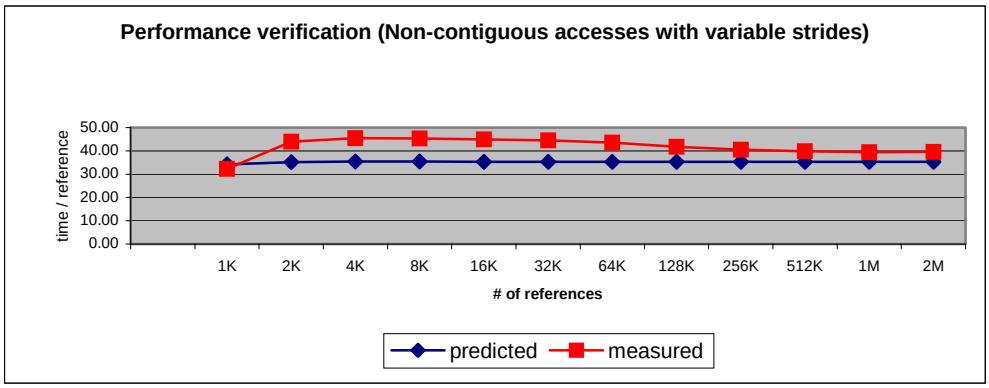


Figure 4.7. Comparison of measured and predicted memory access cost for non-contiguous data access with fixed word size and variable strides (word size: 8 bytes, stride varies from 1 to 128 bytes periodically)

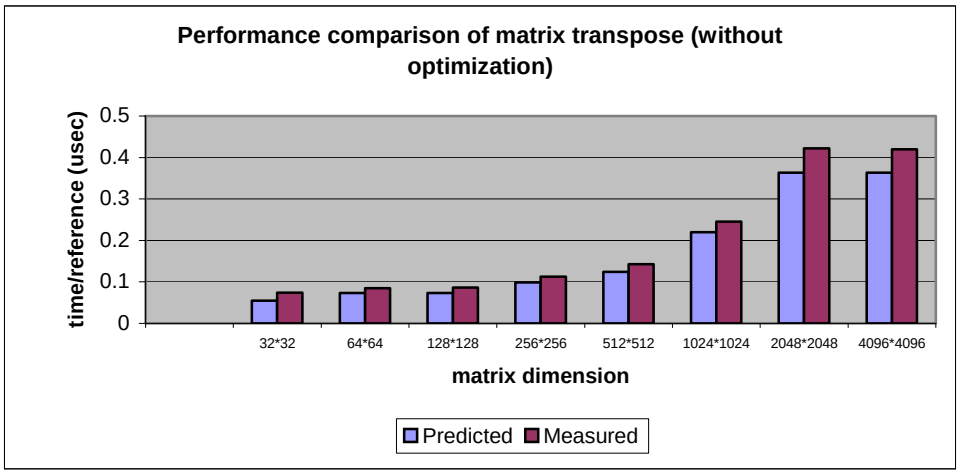


Figure 4.8. Comparison of measured and predicted memory access cost Matrix transpose algorithm without cache blocking optimization

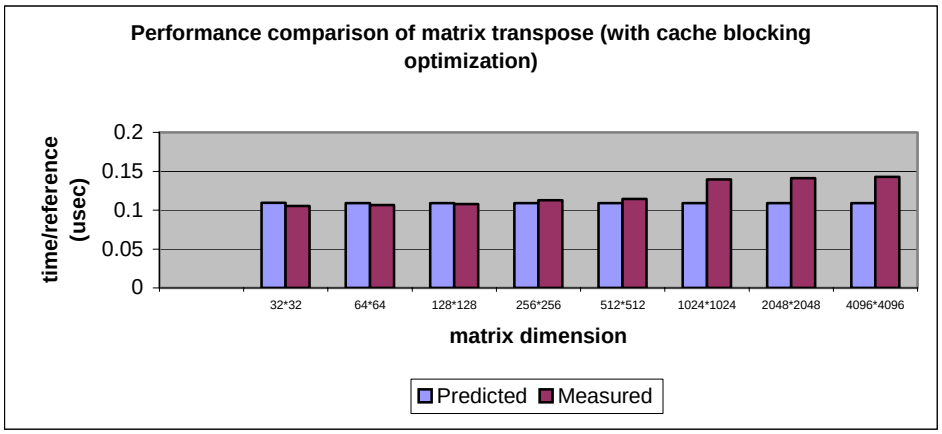


Figure 4.9. Comparison of measured and predicted memory access cost
Matrix transpose algorithm with cache blocking optimization

4.5 UTILIZATION OF OUR MODEL

Our prediction model can be utilized in a broad variety of situations. As we can predict the memory access cost for all access patterns, in the optimization cycle, we can spare repeated source code modification, compilation execution to retrieve performance data. We can just change the access pattern parameters and find the access cost for that pattern. This makes the process of testing many optimized patterns with very small overhead. This model can be directly applied in memory-logP model to find the memory access cost, which is a function of data size and distribution of data. We utilize this mode in improving the performance of MPI derived datatypes [Mpif98], which we explain in the next chapter.

4.6 SUMMARY

In this chapter, we discussed the issues of memory hierarchy and presented our proposed prediction model based on data access patterns. To optimize memory performance, we can use loop transformations and loop access reordering techniques to

improve the memory access performance. To obtain these loop optimization parameters, a simple, fast and accurate memory access cost prediction model is necessary. This improves the standard of application level optimizations and reduces the burden on the programmers to learn the rapidly improving processor and computer architecture technology. Towards achieving this goal, in this chapter we presented an analytical model to predict the memory access cost based on the data access patterns. We first classified the most common data access patterns in scientific computing applications. We then presented a model to predict the memory access cost called SMAC. We verified this model with measurements and showed that this model is practical. The accuracy of our model is reasonable given its simplicity. We also applied this model to matrix transpose routines in Fast Fourier Transform program of NAS benchmarks, which was implemented in different memory access patterns.

Our model is simple, effective, and easy to be incorporated into memory cost tuning tools, where optimization parameters are to be found at runtime. The prediction error of 10% to 20% exists; but our model is reasonably accurate in making optimization decisions. We are currently utilizing this model to improve the performance of MPI derived datatypes, by optimizing the memory access cost (discussed in the next chapter). This cost prediction is a part of our memory-logP model, which emphasizes the importance of memory communication performance in point-to-point communication. Our model is practical because of its simplicity. We are able to fit this easily into any optimization library to choose optimization parameters dynamically at runtime. This is not possible with the existing models due to their complexity.

This model can be extended in various aspects. In future, we plan to extend this model to include external and internal conflict misses. We will broaden this model for replacement policies other than LRU, such as FIFO, LFU, MRU, MFU etc. In future, we plan to incorporate this model in an automatic performance tuning system that improves the application performance by optimizing the memory access cost.

CHAPTER 5

DATA ACCESS OPTIMIZATION FOR MIDDLEWARE

In this chapter, we first present our techniques of utilizing SMAC model for improving the performance of message passing middleware. Communication in parallel applications is a combination of data transfers internally at a source or destination and across the network. Previous research focused on quantifying network transfer costs has indirectly resulted in reduced overall communication cost. Optimized data transfer from source memory to network interface has received less attention. In shared memory systems, such memory-to-memory transfers dominate communication cost. In distributed memory systems, memory-to-network interface transfers grow in significance as processor and network speeds increase at faster rates than memory latency speeds. Our objective is to minimize the cost of internal data transfers. The following examples in this chapter illustrate the impact of memory transfers on communication; we present a methodology for classifying the effects of data size and data distribution on hardware, middleware, and application software performance. This cost is quantified using hardware counter event measurements on the SGI Origin 2000. Our analysis technique identifies the critical data paths in point-to-point communication. For the SGI O2K, we empirically identify the cost caused by just copying data from one buffer to another and the middleware overhead. We use MPICH in our experiments, but our techniques are generally applicable to any communication implementation.

We then explain the methods to improve the performance of MPI derived datatypes utilizing our proposed prediction models. The Message Passing Interface (MPI) has become a de facto standard for parallel programming. This standard supports *derived*

datatypes, which allow users to describe noncontiguous memory layout and communicate noncontiguous data with a single communication function. This feature enables an MPI implementation to optimize the transfer of noncontiguous data. In practice, however, few MPI implementations implement derived datatypes in a way that performs better than what the user can achieve by manually packing data into a contiguous buffer and then calling an MPI function. In this chapter, we present a technique to automatically select templates that are optimized for memory performance based on the access pattern of derived datatypes. We implement this mechanism in the MPICH2 source code. The performance of our implementation is compared to well-written manual packing/unpacking routines and original MPICH2 implementation. We show that performance for various derived datatypes is significantly improved and comparable to that of optimized manual routines.

5.1 MEMORY COMMUNICATION

Computers continue to increase in complexity. Hierarchical memories, superscalar pipelining, and out-of-order execution have improved system performance at the expense of simplicity. Redesigned compilers allow applications to take advantage of new architectures and execute more efficiently. As shown in Chapter 2, compilers are limited in their ability to increase performance. For instance, the compiler is often unaware of subtle characteristics of cache hierarchies. Optimal performance is typically achieved through a combination of optimized compilation and algorithm redesign through system dependent analysis.

Communication in parallel systems increases complexity tremendously. As a result, parallel compilation is even less fruitful than its sequential ancestor. Optimizing performance in such environments relies more heavily on the use of tools to provide performance data for analysis.

Distributed systems rely on middleware to address the interoperability of heterogeneous software and hardware implying additional complexity. The interaction between hardware, software and middleware is not well understood. Thus, the optimizing abilities of distributed compilers are very limited. Optimizing performance in such environments will greatly rely on tools for system dependent analysis.

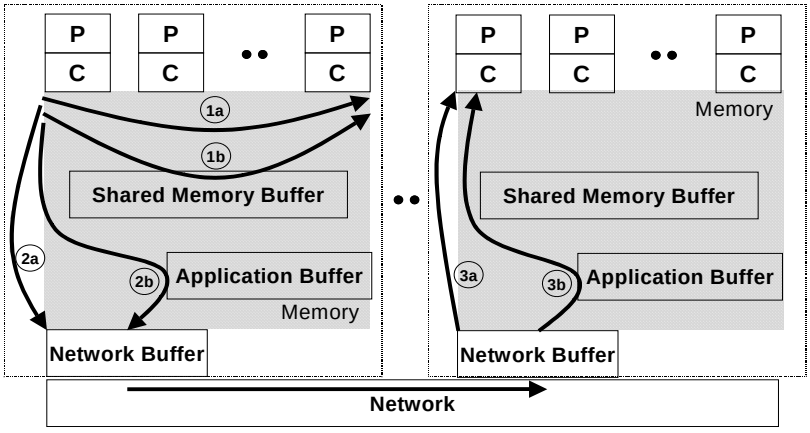


Figure 5.1. Memory communications within shared memory (1a-b) and to/from the network buffer in distributed communication (2a-b/3a-b) follow critical paths dependent upon data size, data distribution, and system implementation.

Communication costs in such environments are a function of the critical data path (see Figure 5.1). A communicated message must be moved from the source's local memory to the target's local memory. *Memory communication* is the transmission of data to/from user space from/to the local network buffer (or shared memory buffer). *Network*

communication is data movement between source and target network buffers. Communication cost consists of the sum of memory and network communication times.

Unfortunately, the same layers of middleware that enable distributed processing (e.g. MPI, PVM) convolute the critical data path. Figure 5.1 illustrates the possible critical paths of data for communication in a simple distributed shared memory machine or cluster. The chosen path (and cost of the communication) depends on the destination, the data size and distribution, and the system implementation of the middleware.

Consider communication in a cluster of shared memory computers. Depending on the underlying communication scheme (e.g. MPI, PVM), the chosen implementation (e.g. MPICH [Mpif98]), and the system architecture design, different communication buffers will be used along the critical data path. For small data sizes, communication proceeds without application-level buffering. For large data sizes, data buffering at the application level will occur. Buffering will also occur at the network level.

The relative cost of memory communication is increasing. Processor speeds continue to outpace memory latency improvements. As the gap widens, memory performance becomes an even larger portion of overall execution time. It is very important to limit the cost of memory accesses. As mentioned, compilers can help, but given the complexity of present and future distributed systems, the first step toward this solution is to identify the critical path of memory communication in a real system and quantify the cost.

In next section, we describe a scientific approach to empirically determine the critical path of memory communication. Our approach [Bycs02] follows traditional methods of characterizing memory hierarchies applied to memory communication. We additionally apply a model of memory communication cost to characterize the effects of locality on

observed buffer transmission [Bycs03, Bycs04]. Our goal is to identify and quantify the costs of memory communication automatically. We test our methodology on a cluster of SMPs to show its usefulness and verify correctness. In the following sections, we describe the experimental setup and then classify memory communication from network communication.

5.2 CLASSIFICATION OF PARALLEL COMMUNICATION

5.2.1 EXPERIMENTAL SETUP

SGI Origin 2000: Distributed Shared-Memory (DSM) multiprocessors provide the convenience of shared memory programming with a scalable design. The SGI Origin 2000 at NCSA utilizes a cc-NUMA architecture running the IRIX version 6.5.14 operating system. The interconnection network for 128 processors is a 5th degree hypercube with 4 processors (2 nodes) per router. High-speed, dedicated Craylink interconnects link nodes. The achievable remote memory bandwidth on Craylink interconnect is 624MB/sec in each direction, which adds a 165ns off-node penalty and 110ns per hop. As long as the communication is between nodes within a hypercube, per-hop latency is zero, but the communication to an outer cube node causes increases in latency.

A directory based tree protocol maintains cache-coherence. A complex memory hierarchy reduces the impact of memory latency. Each node contains two MIPS R10000 processors [Ncsa00]; each running at 195MHz, and 32kB two-way set associative, two-way interleaved primary (L1) cache. An off-chip 4MB secondary unified cache is present as well. Cache and page block sizes are 32 and 4096 bytes respectively. Load misses at L1 and L2 were measured as 12 and 90 cycles respectively. The MIPS R10000 is a four-

way superscalar RISC processor. The machine used in testing has 48 195MHz MIPS R10000 processors, with 14 GB main memory. The available local memory access bandwidth is 680MB/sec in each direction. SGI O2K machine is selected as our platform due to availability and support of hardware counter libraries. As our study focuses on the effect of local memory references, this can be generalized for commodity cluster architectures.

Hardware Performance Counters: All the commodity processors provide hardware performance counters to measure and validate the processor architecture. The MIPS R10000 processor has two on-chip 32-bit registers to count 30 distinct hardware events. In our experiments we have measured the events related to total cycles (event 0), graduated instructions (event 17), memory data loads graduated (event 18), memory data stores graduated (event 19), L1 cache misses (event 25), L2 cache misses (event 26). The overhead of cache misses on SGI Origin 2000 is measured [Moos01] as 1 cycle for register access, 2-3 cycles for an L1 cache hit, 7~13 cycles for an L1 cache miss, 60~200 cycles for an L2 cache miss. A TLB miss costs more than 2000 cycles. This shows that the overhead increases massively as L2 cache misses occur. We have chosen these counters to study the memory effect on communication as they reflect the memory operations for any processor.

Performance Measurement: We measure each experiment twenty times for all data sizes and hardware counter events. Accuracy is maintained by taking only the values with low standard deviation. In the next section, we quantify performance degradation for non-contiguity of the data using the memory-logP model [Casu03]. For this, we use a program that measures performance for sending a contiguous and a non-contiguous message with

point-to-point blocking message passing under various data sizes and strides. We study point-to-point, blocking communication due to its prevalence in parallel applications. However, our methodology applies to any middleware implementation. In the third experiment, we use a matrix transpose algorithm to represent memory communication packing and unpacking. We measure the costs for simple contiguous message copying, non-contiguous message copying, and sending data to a remote processor.

Communication cost for sending a data segment depends on architectural parameters (e.g. cache capacity) and code characteristics (e.g. data distribution) as explained in the memory-logP model [Casu03]. Typically, a message transmission involves data collection, data copying to the network buffer and data forwarding to the receiver. When a data distribution is not contiguous, typically it is collected into a contiguous buffer before copying to the network buffer (see Figure 5.1). This intermediate copying is costly as data sizes and strides increase resulting in additional capacity and conflict misses to the cache [Larw91]. This can be done without extra buffer copying by directly copying to the network buffer. However, the performance degrades further due to poor utilization of network buffer. Strided accesses decrease the efficiency of cache hierarchies designed to exploit locality (capacity misses). Caches with less than full associativity, often a small power of 2, suffer from mapping collisions under certain access patterns (conflict misses).

The parameters of the memory logP model capture architecture and code characteristics. Memory cost of transferring data of a specific size is a combination of unavoidable overhead (o), and effective latency (l) – a function of data size and distribution. Additional network latency after removing overlap exists in passing a

message between two processors. Total communication time increases with memory latency (*l*). In our micro benchmark experiments, memory performance worsens with an increase in stride. We use the memory-logP model to enumerate the memory hierarchy performance so that a developer can improve the performance of those parts of code with locality-conscious optimizations. Quantifying the memory communication costs is the first step in bottleneck identification. Succeeding analysis can identify system buffer-related parameters.

Figures 5.2 and 5.3 illustrate the communication cost and cause of 16-Dword (16 x 8 bytes), strided data transfers using MPI Blocking Sends. The contribution of memory communication to total communication is obvious. As message sizes increase for fixed strides, data transfer time increases (Figure 5.2) from additional conflict and capacity misses (Figure 5.3) in the memory communication. The rate of cost increase is dependent upon the data distribution and the memory hierarchy characteristics.

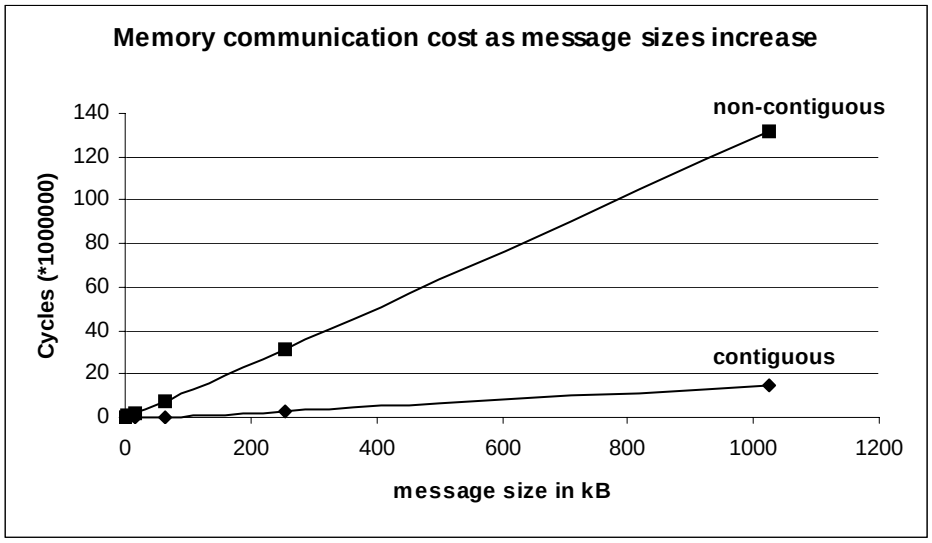


Figure 5.2. Total cost for sending contiguous (stride 1) and non-contiguous (stride 16) messages. The costs are similar at low data sizes and it increases a lot once the data does not fit into the cache or when TLB thrashing occurs.

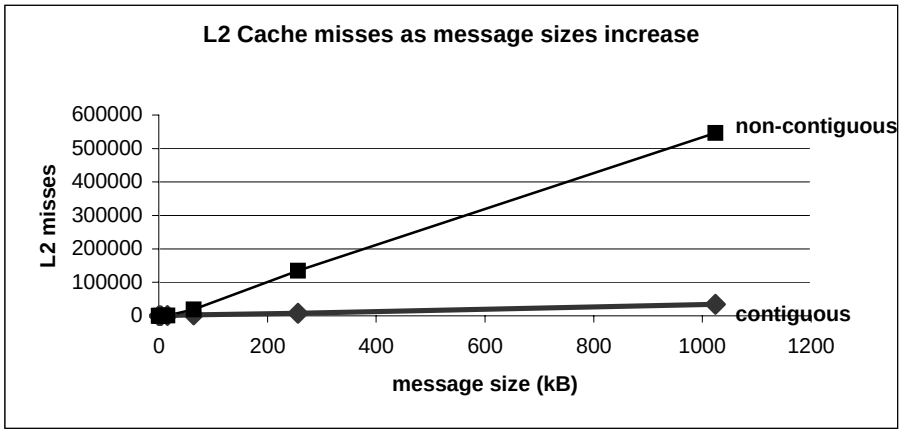


Figure 5.3. L2 cache misses for sending contiguous (stride 1) and non-contiguous (stride 16) messages. Each L2 cache miss costs between 60 to 200 cycles.

Estimation of the o parameter of the memory logP model requires measurement of contiguous data transmission, a relatively simple task using micro-benchmarking techniques. We expect the o parameter increases proportionately as problem size and strides increase; that a scalable transmission method is chosen. Recent work [Wogr02] indicates that the overheads for packing and unpacking of MPI derived data types in implementations such as MPICH do not scale well.

Measuring the *l* parameter directly requires running experiments varying message size and contiguity. After subtracting the ideal overhead, the *l* function remains. In Figure 5.4, a comparison of various costs is shown. The first bar in each grouping quantifies overhead (*o*) of copying a contiguous data block. In the SGI O2K system measured, overhead remains constant as size increases. The second bar in each grouping shows the cost for packing a non-contiguous message into an intermediate buffer. This cost is similar to copying a contiguous message when the data fits totally into the cache. After the data size crosses this barrier, the costs increase due to cache and TLB misses. Sending a contiguous message includes a small fraction of memory copying cost (*o*) and network

latency, but for non-contiguous messages, this memory cost increases with the data size as it includes the latency parameter (*l*).

5.3 IDENTIFYING MEMORY COMMUNICATION

Memory hierarchies are complex. System middleware (e.g. MPICH) provides abstractions (e.g. derived data types) to simplify distributed programming hiding the details of data transfer from the user. Determining the particular costs of memory communication is non-trivial due to the complex interaction between application, middleware, and hardware. However, to optimize performance, application developers must understand the full cost of communication. Using the quantifiable parameters of the

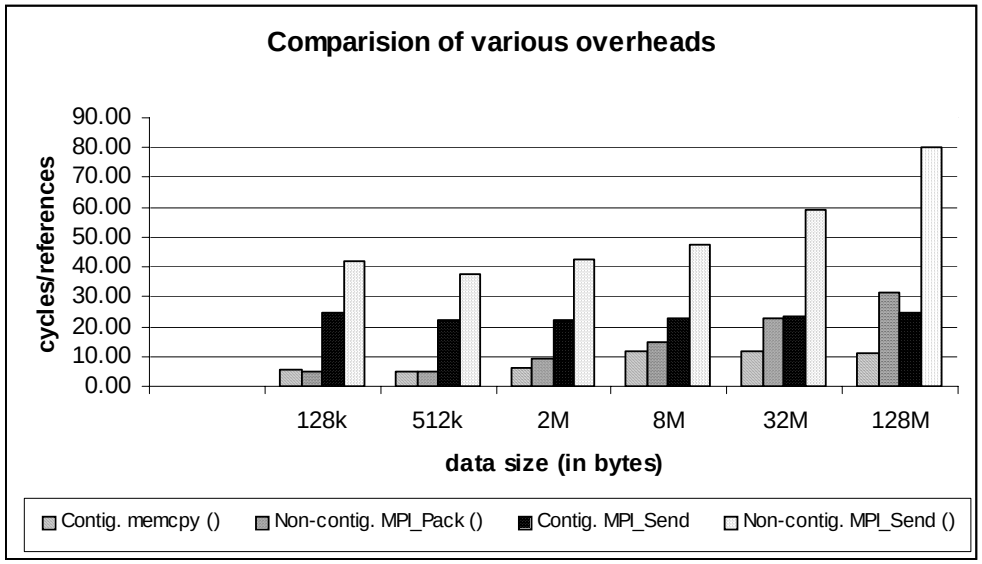


Figure 5.4. Comparison of cost for various implementations of transpose algorithm. Contig.memcpy () : Copying data from one buffer to another using memcpy (). This is the basic overhead (o) to copy contiguous data. Non-contig. MPI_Pack () : This packs columns of matrix using MPI_Type_vector (). This cost is a combination of (o) and (l). Contig.MPI_Send () sends a contiguous message over network to another processor. This includes the cost of small contiguous copying overhead (o) and the cost for network transfer and software overhead of MPI_Send. Non-contig. MPI_Send () packs a non-contiguous data to transpose and sends to the receiver. This cost includes the packing overhead, copying data from memory to the network buffer and the network cost.

memory logP model and micro-benchmark experiments, it is possible to identify buffer copies in shared memory architectures.

Specifically, we test various data sizes and strides iteratively and observe the largest gap among the successive hardware counter values after consideration of experimental variation. These gaps or significant changes pinpoint policy decisions in the case of MPI codes (application buffers) and memory hierarchy characteristics (implicit buffering). At the memory hierarchy level, our approach is similar to that of traditional micro-benchmarking techniques [Sagc93, Whal01] used to identify general cache characteristics. We additionally verify our analyses with hardware counter data; this is particularly important for identifying application-level buffers.

Inefficient memory communication is not limited to exploitation of the memory hierarchy. Figure 5.4 shows the increases in the latency parameter (l) with additional layers of overhead caused by middleware such as MPICH implementation is measured. It has been our experience that code developers targeting performance generally avoid certain abstractions such as derived data types since they understand the overhead resulting from such abstraction negatively and significantly impacts performance. Figure 5.4 affirms this intuition. Figure 5.5 shows the classification of various costs including (l), (o) and other latency. Collective costs of middleware are very high with the increase of message size. This depicts that the magnitude of the cost differential is truly system and application dependent. Hence, a more scientific approach to determine when to use abstractions such as derived data types would involve determining the exact cost for a specific application-architecture combination. This is the purpose of our techniques and the original motivation behind this work.

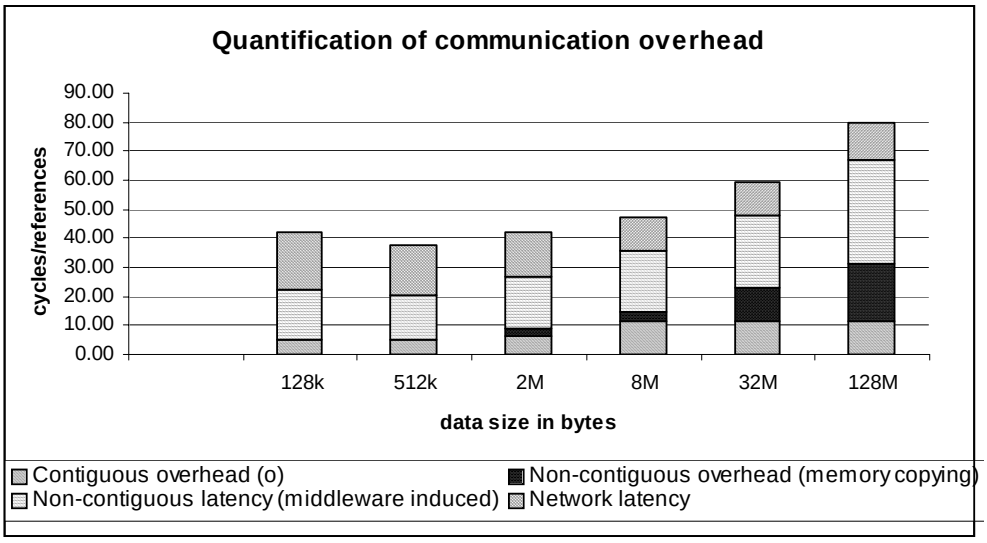


Figure 5.5. From figure 5.4, Contig. memcpy () = o, MPI_Send () contains the overhead due to copying of data from memory to the network buffer and network latency. Non-contig MPI_Send () has the additional buffer copying cost over MPI_Send () costs.

As discussed, the critical data path for communicating a message varies depending on the destination, data size, and distribution and the implementation of middleware (see Figure 5.1). For example, MPI [Mpif98] uses three protocols in buffering: short, eager and rendezvous. In “short” protocol, data is sent with the envelope of the message. This protocol is ideal for very small data sizes. With “eager” protocol, data is not stored in an application buffer at the sender, assuming the receiver can store the data (see Figure 5.1 2a/3a-b). This method is advantageous to reduce any synchronization delays. But buffering may require additional space for messages from an arbitrary number of senders. Memory exhaustion may occur at the receiver for large data. In “rendezvous” protocol, data is buffered at the sender until the receiver responds and posts a receive signal. This scheme provides scalability but comprises extra handshaking delays and stall for the cases of buffer exhaustion.

A standard-mode send function uses “eager” protocol while the message size is less than the available system buffer size. After the buffer size is surpassed, the system switches to rendezvous protocol. There are many implementations to deal with non-contiguous data. In a typical MPICH implementation [Mpif98], buffering is needed regardless of message size (Figure 5.1 *2b*) for strided datatypes. At the receiver, unpacked data requires an additional copy (Figure 5.1 3a); when unpacking is not necessary, no additional copy is made (Figure 5.1 3b). The CPU stalls when the buffer exhausts until the data is sent out of the buffer. Location of the receiver also impacts the overall communication cost in shared memory systems.

Comparing the performance of communication for contiguous and strided messages can isolate the overhead caused by additional buffering. The cost of sending a contiguous message between two processors is a combination of data transfer overhead (*o*) and network latency. Sending a strided message has extra overhead due to poor exploitation of memory hierarchy and additional buffer copying. Optimally, the cost of copying strided data into a contiguous buffer is the same as the cost of packing it using MPI implementation. Additional middleware induced overhead is separated by subtracting the costs of sending contiguous message and that of packing from the total cost of sending strided message. Figure 5.5 depicts the partition of these costs. This is an empirical method of separating all the costs in memory communication. Presentation of these costs provides a developer with an insight into exploiting the advanced memory hierarchies, and to decide which critical data path is optimal to use.

5.4 MPI DERIVED DATATYPES

The MPI (Message Passing Interface) Standard is widely used in parallel computing for writing distributed-memory parallel programs [Mpif95, Mpif98, Grla99]. MPI has a number of features that provide both convenience and high performance. One of the important features is the concept of derived datatypes. Derived datatypes enable users to describe noncontiguous memory layouts compactly and to use this compact representation in MPI communication functions. Derived datatypes also enable an MPI implementation to optimize the transfer of noncontiguous data. For example, if the underlying communication mechanism supports noncontiguous data transfers, the MPI implementation can communicate the data directly without packing it into a contiguous buffer. In contrast, if packing into a contiguous buffer is necessary, the MPI implementation can pack the data and send it contiguously. In practice, however, many MPI implementations perform poorly with derived datatypes—to the extent that users often resort to packing the data manually into a contiguous buffer and then calling MPI. Such usage clearly defeats the purpose of having derived datatypes in the MPI Standard. Since noncontiguous communication occurs commonly in many applications (for example, fast Fourier transform, array redistribution, and finite-element codes), improving the performance of derived datatypes has significant value.

The performance of derived datatypes can be improved in several ways. Researchers have used data structures that allow a stack-based approach to parsing a datatype, rather than making recursive function calls, which are expensive [Thrz99, Grld99, Romg03]. These works improved the performance of derived datatypes to the level of performance with naïve manual implementations for packing noncontiguous data. (We do better than

that with our optimizations.) Wu et al. [Wuwp04] improved the performance of MPI derived datatypes by taking advantage of the features in InfiniBand to overlap packing and unpacking a message with network communication.

The performance of derived datatypes can be improved further by using optimized algorithms for packing and unpacking of data. Many implementations of derived datatypes use loops in packing/unpacking noncontiguous data. Utilizing data locality in these loops by applying loop optimizations, which a developer cannot easily do without advanced knowledge of memory hierarchy design and optimizations, is beneficial. This area is the focus of our research. These techniques are useful for MPI implementations on various network channels and the performance gain is not limited to fast networks. To our knowledge, no other MPI implementations use memory-optimization techniques for packing noncontiguous data in their derived-datatype code (for example, see the results with IBM's MPI in Figure 5.12).

Much research has been performed by the compiler and algorithms community on improving memory-hierarchy performance by using techniques such as loop transformation, array padding, and cache blocking [Larw91, Kand99, Rivt99], and we use some of these optimizations in our optimized packing algorithms. Many compilers use some of these optimizations to improve code performance. However, longer compile times and dynamic behavior of data accesses limit the performance improvement that a compiler can obtain. Many software libraries have been developed, particularly for numerical software, that use advanced memory-optimization techniques, for example, the portable LAPACK library [Weblpk] and the ESSL and PESSL libraries on IBM machines [Webibm]. ATLAS (Automatically Tuned Linear Algebra Software) is an

approach for automatically generating optimized numerical software on machines with deep memory hierarchies [Whal01].

As explained in earlier sections, inter-process communication (IPC) can be considered as a combination of *memory communication* and *network communication*. Memory communication (or memory copying) is the transfer of data from the user's buffer to the local network buffer (or shared-memory buffer) and vice versa. Network communication is the movement of data between source and destination network buffers. Limiting the cost of memory accesses can significantly improve the overall communication time, as we demonstrate in this chapter. The key to improving the memory-access performance is to exploit advanced memory hierarchies in modern computer architectures. Doing so directly is difficult for users because they are often unaware of architectural details, and these details vary from machine to machine. We do the memory optimizations automatically at the library level in our memory-conscious implementation of derived datatypes.

We first implement our optimizations in MPICH-1 version, where we present the scope of performance improvement by using MPI's profiling interface (PMPI). In this chapter, we present automatic selection of optimized packing/unpacking templates within the MPICH2 source code, based on data access patterns, data size, and memory architecture. Ogawa et al. [Ogma96] used optimized templates in improving MPI performance for instantiating partial-evaluation code selection in order to reduce software overhead. We, in contrast, use templates to optimize memory performance.

For MPICH-1 optimization at profiling MPI level, except for the results in Figure 5.12 with IBM's MPI, we performed experiments on an SGI Origin2000 at the National

Center for Supercomputing Applications. This machine has a cc-NUMA architecture and runs IRIX 6.5.14 as the operating system. Each node of the machine has two MIPS R10000 processors [Ncsa00] running at 195 MHz. Each processor has a 32 KB two-way set-associative and two-way interleaved primary (L1) cache and a 4 MB off-chip secondary cache. The MIPS R10000 processor has two on-chip 32-bit registers to count 30 distinct hardware events. In our experiments, we measured the events related to total cycles (event 0), graduated instructions (event 17), memory data loads graduated (event 18), memory data stores graduated (event 19), L1 cache misses (event 25), and L2 cache misses (event 26). The MPI implementation we used is MPICH-1.2.5 with the shared-memory device.

For optimizations in MPICH2 source code, to test the portability of our optimized implementations, we ran these experiments on two different clusters: a 350-node Linux cluster (*jazz*) at Argonne National Laboratory and an 84-node Sun cluster (*sunwulf*) at Illinois Institute of Technology. The nodes of *jazz* have a 2.4 GHz Pentium-4 processor with 1 GB of memory. These processors have 512 KB of built-in L2 cache, with a 64 byte cache line and 8-way associative, a TLB of 128 entries, and a page size of 4 KB. These nodes are connected with Fast Ethernet. Each node of the *sunwulf* cluster is a Sun Blade-100 workstation with one 500MHz UltraSparc-IIe CPU. The L1 cache is 16 KB, with a 16-byte cache line size. The L2 cache has a capacity of 8 MB and its line size is 64 bytes. It has a TLB of 48 entries with 4 KB page size. These nodes are connected with Gigabit Ethernet.

5.5 OPTIMIZING MPI DERIVE DATATYPES

In this section, we describe how we automatically use memory-optimized packing algorithms to implement MPI derived datatypes [Bgst03, Bgst031].

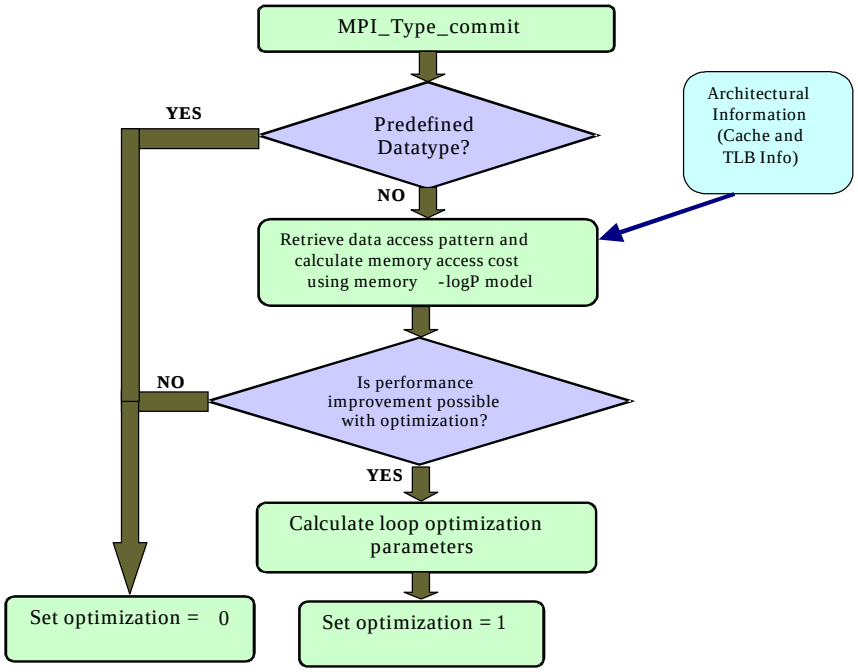


Figure 5.6. Predicting the performance of memory access cost and improving the performance of MPI derived datatypes

Overview of optimization technique: Figures 5.6 and 5.7 illustrate the procedure for optimizing sends with derived datatypes. The profiling interface in MPI [Mpif95] provides a convenient way for us to insert our code in an implementation-independent fashion as follows. Every function in MPI is available under two names, MPI_ and PMPI_. User programs use the MPI_ version of the function, for example, MPI_Send. We intercept the user's call to MPI_Send by implementing our own MPI_Send function in which we do the datatype packing (if necessary) and then call PMPI_Send to do the data communication. As a result, application programs don't need to be recompiled; they need only to be re-linked with our version of the MPI functions appearing before the MPI

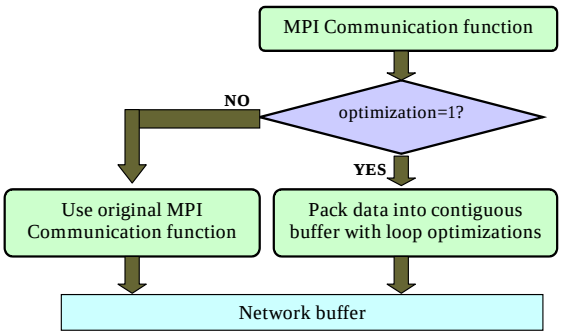


Figure 5.7. Packing the data to improve the performance of MPI derived datatypes

library in the link command line. We currently interpret derived datatypes by accessing the internal data structures used by MPICH. However, we plan to adopt an implementation-independent way by using the datatype-decoder functions from MPI-2, namely, MPI_Type_get_envelope and MPI_Type_get_contents. In our implementation of MPI_Send, we first determine whether the datatype passed is a basic (predefined) datatype or a derived datatype. If it is a basic datatype, we simply call PMPI_Send as no packing is needed. For a derived datatype, we first determine whether any performance improvement is possible for the access pattern represented by the datatype as described below.

Memory performance degrades significantly when the program cannot reuse the data, which is already loaded in various levels of the memory hierarchy. We observed in Figure 5.2 that the data-packing overhead (l_p) increases after data size 2 MB. Cache reuse degrades when the number of cache lines required to be loaded for the working set of data is more than the available number of cache lines. Reuse of the Translation Look-aside Buffer (TLB) degrades when the number of page entries required is more than the number of entries the TLB can hold. The TLB typically contains a small number of entries, which result in more misses when the number of pages required is higher than

this number. Therefore, to determine whether any performance improvement is possible, we use the metric of determining whether there will be any TLB misses for a naïve (unoptimized) data-packing algorithm for the given data size. For this purpose, we need to know the number of page entries that will be required to be loaded into the TLB. For a noncontiguous access with fixed block size and fixed stride, this number can be determined as follows.

Let W be the size in bytes of each contiguous block, S be the stride in bytes between the start of two consecutive contiguous blocks of data, n be the number of references to contiguous blocks of data in the innermost loop, Ps be the page size in bytes, and Tp be the maximum number of entries the fully-associative TLB can hold. If $P_s \geq S$, each page contains $\lfloor \frac{P_s}{S} \rfloor$ references. The number of pages required R_p can be calculated as follows:

$$R_p = \frac{n}{\lfloor \frac{P_s}{S} \rfloor} \text{ if } P_s \geq S, \text{ and } R_p = n \left\lceil W / P_s \right\rceil \text{ if } P_s < S.$$

If $R_p < T_p$, the entire data to be accessed, including the stride, can be mapped by TLB; that is, there are no TLB misses. In this case, we assume that no performance optimization is possible, and we simply call PMPI_Send. But if $R_p \geq T_p$ and if the access pattern represented by the datatype includes out-of-order accesses, some of the pages mapped would be replaced before they are completely used. In this case, we use our optimized packing algorithm that uses cache blocking to ensure better reuse of mapped pages.

Choosing a Block Size: After determining that performance improvement is achievable, performance-optimization parameters (such as block size for cache blocking)

are calculated. Lam et al. [Larw91] have shown that the block size has a significant effect on blocking algorithms. Choosing the optimal block size, however, is difficult. Temam et al. [Tefj98] propose an analytical approach to calculate block size by taking into consideration all three types of cache misses: compulsory, capacity, and conflict misses. This method has significant overhead in estimating the parameters accurately.

Instead of trying to find the optimal block size, we simply aim to choose a block size that avoids the worst performance. Specifically, because of the high cost of TLB misses [Safs00] and the relatively small size of the TLB, we decided to choose a block size that minimizes TLB misses. In our implementation, we choose a block size that accommodates TLB mapping, noncontiguous array accesses, and the other variables in the program. We use half the TLB entries ($T_p / 2$) to map the block and the other half to accommodate the contiguous buffer and other loop variables. In other words, we use a block size that will consume half the entries in the TLB. We determine the TLB size for a given system by running a microbenchmark developed by Saavedra et al. [Sagc93]. The page size is determined by using the system command getpagesize ().

Choosing a Packing Algorithm: Based on the data-access pattern represented by a datatype, we choose a predefined packing function and plug in the memory-optimization parameters. To select a function, we classify access patterns into combinations defined by contiguous or noncontiguous accesses with fixed or variable block sizes and fixed or variable strides. For each of these combinations we use a predetermined packing function with architecture-dependent parameters. The data is packed into a contiguous buffer with the selected packing function, and we then call PMPI_Send with the contiguous buffer.

Figure 5.8 shows the current implementation of MPI_Send in MPICH, and Figure 5.9 shows our implementation with the packing optimizations.

```
MPI_Send (data, datatype, dest)
{
      if (datatype is basic datatype)
      {
            Send (data) to the network buffer.
      }
      else (datatype is derived datatype)
      {
            /* MPI_Pack () cost is staggeringly high for large
               data sets and powers-of-2 dimension arrays */

            MPI_Pack (data, datatype, buffer);
            Send (buffer) to the network buffer.
      }
}
```

Figure 5.8. Current implementation of MPI_Send in MPICH-1

To directly apply the optimizations in the source code of MPICH2 [Bstg06] instead of PMPI [Bsgt03], we developed a systematic approach. Our method first retrieves the data access pattern of a derived datatype from user's definition and verifies whether performance improvement is possible with optimizations for a derived datatype before applying them, as shown in Figure 5.6. If improvement is possible, our optimization method uses an analytical model [Bsgt04] to predict memory access cost and to find optimization parameters with the lowest access cost. These parameters are passed to templates to pack/unpack noncontiguous data.

Overall procedure of optimizing an MPI communication function using derived datatypes has two steps. In the first step, we verify whether a datatype is optimizable or not, and find optimization parameters. In the second step, MPI communication function calls optimized templates automatically.

```
MPI_Send (data, datatype, dest)
{
       if (datatype is basic datatype)
       {
              PMPI_Send (data, datatype, source, dest);
       }
       else (datatype is derived datatpe)
       {
              packing_algorithm = Select_best_packing_algorithm (data,
              datatype);
              pack (packing_algorithm, data, datatype, buffer) ;
              PMPI_Send (data, datatype, dest);
       }
}

Select_best_packing_algorithm (data, datatype)
{
       if (data fits into cache/TLB)
       {
              packing_algorithm = PMPI_Pack (data, datatype);
       }
       else
       {
              calculate_optimization_params (datatype, system_info, &params);
              choose_packing_algorithm (params, data, datatype,
              &packing_algorithm);
       }
       return (packing_algorithm);
}

pack (packing_algorithm, data, datatype, buffer)
{
       if (packing_algorithm == PMPI_Pack)
       {
              PMPI_Pack (data, datatype, buffer);
       }
       else
       {
              /* Here come the template implementations for various
                 data-access patterns with optimized parameters */
       }
}
```

Figure 5.9. Memory-conscious implementation of MPI_Send

In MPI programs, after defining a derived datatype, it has to be committed by calling `MPI_Type_commit`. We modified the implementation of the `MPI_Type_commit` function to verify whether optimization is possible. The modified implementation first retrieves the data access pattern, which includes the type of the user-defined datatype, old datatype, strides between consecutive memory accesses, size of the data items, and depth of the derived datatype. If the old datatype is another derived datatype (that is, when a

derived datatype is nested), `MPI_Type_commit` retrieves these values for that inner datatype as well. We use the datatype decoder functions of MPI-2, namely `MPI_Type_get_envelope` and `MPI_Type_get_contents` to retrieve the pattern. The overhead of decoding datatypes by using these functions is low.

In order to determine whether a datatype is optimizable or not, the modified `MPI_Type_commit` function verifies a series of heuristics that cause cache misses. It verifies whether the datatype is contiguous or noncontiguous, examines whether the data size is more than cache size, and then calculates the factor of cache and TLB reuse. The optimization method reverts back to the original implementation if it determines that the performance cannot be improved at any of these verifications. We use an optimization flag (`is_optimizable`) to keep track of the results of these verifications. If the performance can be improved, `MPI_Type_commit` determines the optimization parameters and sets the flag `is_optimizable` to 1.

We developed optimized templates to pack/unpack noncontiguous data by using various loop optimization methods. In our current implementation, these templates use cache blocking [Larw91], loop unrolling, array-padding optimizations, and software-level prefetching [Mogu91].

In finding optimization parameters to pass to the templates, we first select these optimization parameters based on heuristics explained earlier. To determine if these parameters are optimal or not, we use SMAC prediction model [Bsgt04]. This model verifies whether the memory access cost is reduced with the selected parameters. A new set of optimization parameters are selected if the cost is not optimized and the prediction model verifies for lowered cost again.

For software prefetching, the number of loop iterations needed to overlap a prefetching memory access is called the *prefetching distance* [Molg92]. Assuming memory access latency is *l*, and the work per loop iteration is *w*, the prefetch distance is *ceiling (l/w)*. The main loop that packs data is unrolled for all the references that reuse cache lines that are prefetched. An *epilogue loop* is called without prefetching to execute the last few iterations that do not fit in the main loop. We use a special `gcc` function `__builtin_prefetch` to issue these prefetch instructions. A special flag, `–mcpu,` has to be set to compile MPI source code.

In the second step, when the `MPI_Send` function is called to send the data, if the `is_optimization` flag is 1, the `MPI_Send` calls optimized packing templates using the optimization parameters. These templates are also used when the user calls `MPI_Pack` or `MPI_Unpack` to pack or unpack noncontiguous data.

5.6 PERFROAMANCE EVALUATION

We present the performance results with our implementations at PMPI level on MPICH-1 source code, and then the optimizations performed directly in MPICH2 source code. The first set of results demonstrates the potential of data access performance optimization on derived datatypes of MPI. The second set of results shows the actual performance improvement for various datatypes, including vector and indexed datatypes as well as MG and transpose applications of NAS parallel benchmarks.

5.6.1 MPICH-1 PMPI OPTIMIZATION RESULTS

This section presents performance results for the matrix-transpose example. We compare the performance of three cases: original MPICH with derived datatypes, original

MPICH with manual (unoptimized) packing by the user (no derived datatypes), and derived datatypes with our optimized packing algorithm. To describe the transpose operation with a derived datatype, we use a datatype that is a vector of vectors (vectors of columns in an array). We use a ping-pong operation to measure performance. A process sends a message with a derived datatype representing the transpose, and the destination process receives it contiguously. The destination process then sends back the data with the same derived datatype to the first process, which receives it contiguously. The time is measured at the first process and halved to find the communication cost for one complete data transfer. We run a many iterations of the program and find the minimum time.

In the optimized packing algorithm, cache blocking is used only if the number of pages required to be loaded in the TLB is more than the available TLB entries. For the MIPS R10000 processor on the Origin2000, the number of TLB (fully-associative) entries is 64, and the page size is 16 KB. For matrix transpose, the number of pages required is more than the available TLB entries for arrays of size larger than 512*512 double-precision numbers (data size is 2 MB).

Figure 5.10 shows the performance of the three cases in terms of the number of clock cycles per memory reference. For small data sizes, where we do not use cache blocking, the performance of the three methods is almost the same. But once the data size is 8 MB or larger, where cache blocking comes into effect, our optimized implementation significantly outperforms both original MPICH and manual packing, and the performance improvement is greater as the data size increases.

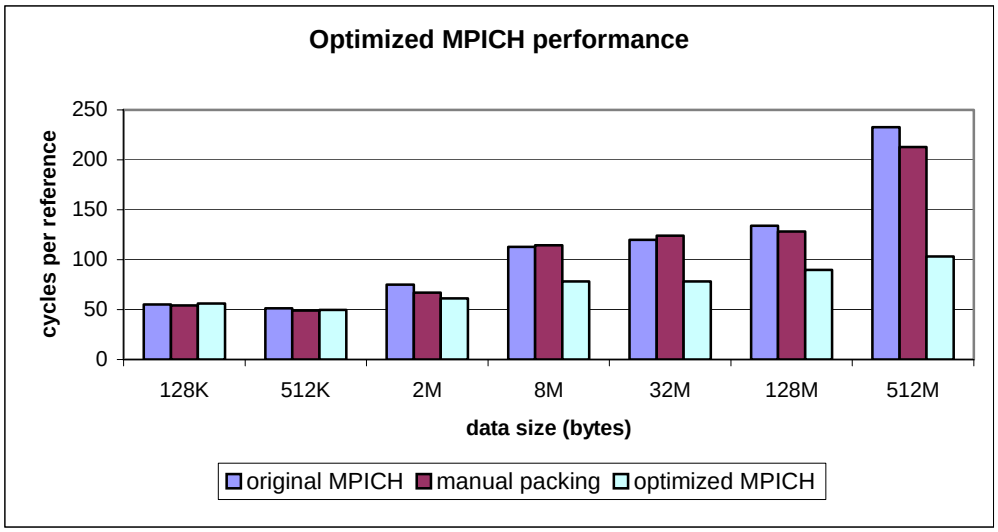


Figure 5.10. Performance improvement with optimized implementation of derived datatypes

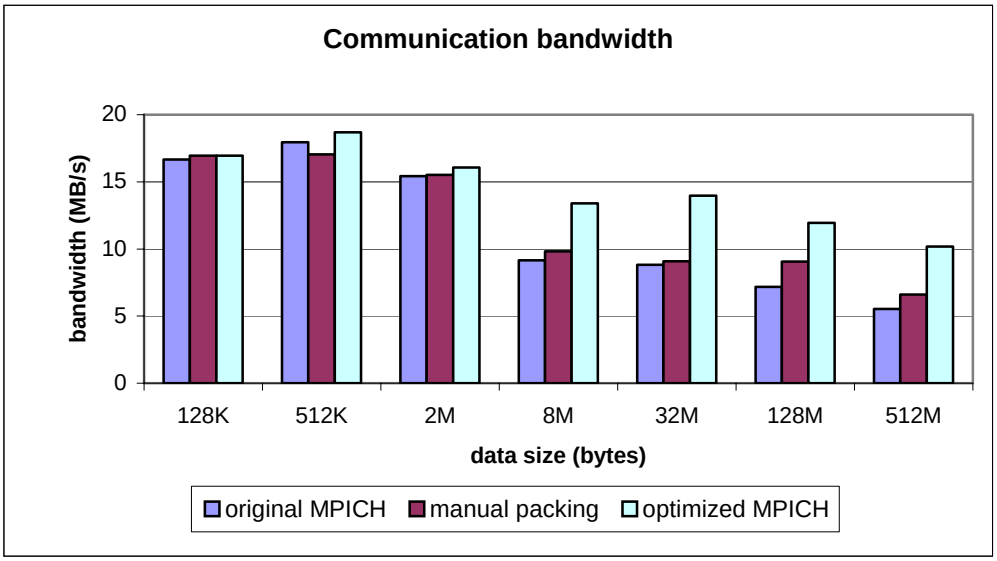


Figure 5.11. Bandwidth improvement with the optimized implementation

Figure 5.11 shows the overall communication bandwidth achieved for the same example with original MPICH and the memory-optimized version. For higher data sizes, the bandwidth achieved by original MPICH decreases considerably, whereas the bandwidth with the memory-optimized version decreases only slightly and is as much as 85% higher than the original MPICH bandwidth.

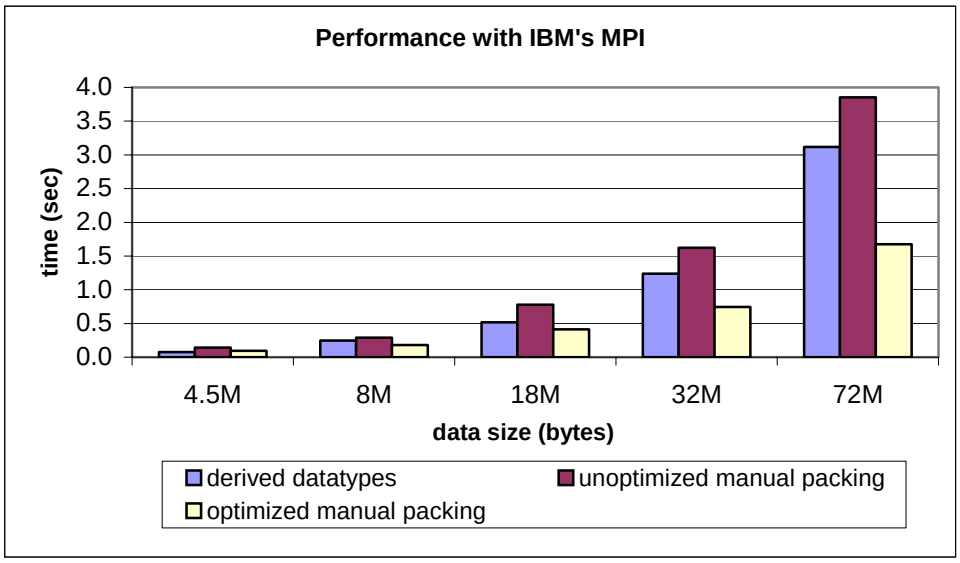


Figure 5.12. Performance of matrix transpose with IBM's MPI

To see how memory-optimized packing performs compared with derived datatypes in a vendor MPI implementation, we ran some experiments on the IBM SP at the San Diego Supercomputer Center. We used IBM's MPI to run three versions of the matrix-transpose program: derived datatypes, manual unoptimized packing, and manual memory-optimized packing. The results are shown in Figure 5.12. For large data sizes, the program that uses manual memory-optimized packing takes significantly lower time than both the derived-datatypes version and the one with manual unoptimized packing. These results demonstrate that vendor MPI implementations also stand to gain by using memory-optimized packing.

5.6.2 MPICH-2 OPTIMIZATION RESULTS

We used three sets of benchmarks to evaluate the performance of our optimized implementations.

1. Simple derived datatypes: We chose fixed derived datatypes defined by the SKaMPI benchmark [Reth00]. They describe a memory layout consisting of a number of units of a basic datatype. The number of units depends on the size of data, the size of basic datatype, and strides. We used vector and indexed datatypes.

2. Nested derived datatypes: We use the nested derived datatypes described by Ross et al. in [Romg03]. These datatypes represent a collection of elements from a 3D array. When a 3D array is stored in row-major order, accessing the YZ face and all the YZ faces of the array in X direction is noncontiguous and has poor locality when the size of the YZ face is more than the cache or TLB sizes. We tested a nested datatype describing a 3D cube of YZ planes in the X direction with a vector of vectors (vector of YZ planes in an array).

3. NAS benchmarks: Lu et al. [Lwps04] modified four NAS benchmarks to apply MPI derived datatypes for noncontiguous data communication. Among these, LU, BT, and SP have small data transfers and do not benefit from memory optimizations. In the MG benchmark, the data transfers in the `comm3` function are noncontiguous and are implemented as packing-then-sent by a sender process and receive-then-unpacking by a receiver. The datatypes described in the modified code are nested datatypes that represent vectors of vectors. We also tested the performance of the matrix transpose operation from the NAS parallel benchmarks' Fourier Transform (FT) program, using MPI derived datatypes. To describe the transpose operation with a derived datatype, we use a datatype that is a vector of vectors (vector of columns in an array).

Except for the NAS MG benchmark, we obtained the performance results of all other benchmarks with an `MPI_Send/Recv` ping-pong operation. In this operation, a process

sends a noncontiguous message that is described by the MPI derived datatypes, and a destination process receives it contiguously. The destination process then sends back the data with the same derived datatype and is received at the first process contiguously. The time is measured at the first process and halved to find the communication cost for one complete data transfer. We ran 20 iterations of each program and calculated the minimum time. We present the performance as transfer rate (MB/s) to normalize the results. The size of the message used in the ping-pong operation is divided by the measured time to find the rate. For the NAS MG benchmark, we compare the execution time of the benchmark.

We compare the performance results for three implementations: manually packing data and sending it (no derived datatypes), MPICH2 version 1.0.3 (unoptimized), and our optimized implementation of the MPICH2 code. The manually implemented pack and unpack codes are written to represent the way a good programmer would write them. Ross et al. [Romg03] showed that the implementation of derived datatypes in MPICH2 outperform those implemented in LAM/MPI [Lamm06]. Therefore, we directly compare our results with MPICH2. We compile all manual codes and MPI installations with `gcc` version 3.2.3 with the flags `-O6`.

Figure 5.13 shows the performance (rate of sending/receiving data in MB/s) of programs using messages formed by vector and indexed datatypes on the *jazz* cluster. Figure 5.14 shows the performance of the same programs on the *sunwulf* cluster. On both clusters, when the message size is larger than cache size, the performance of the original MPICH2 implementation degrades sharply compared to the manual implementation for both vector and indexed datatypes. With the optimized implementation, this performance

is in the same level as that of optimized manual codes. These figures also show that the overhead of optimized implementations is low.

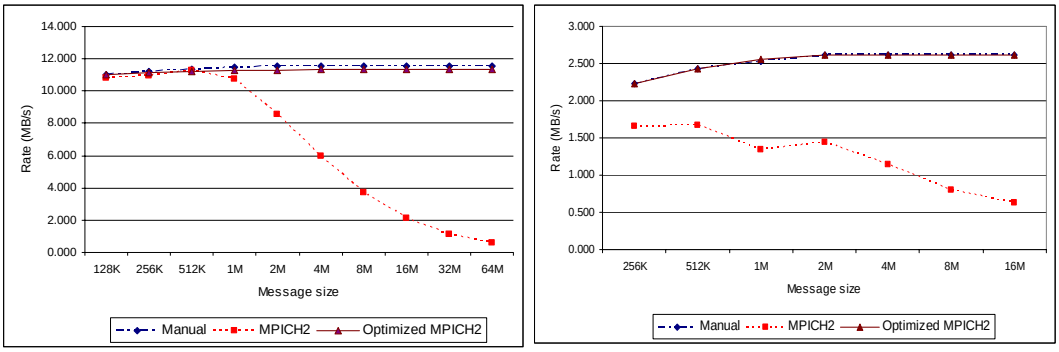


Figure 5.13. Bandwidth measurements for vector (left) and indexed (right) datatype on jazz

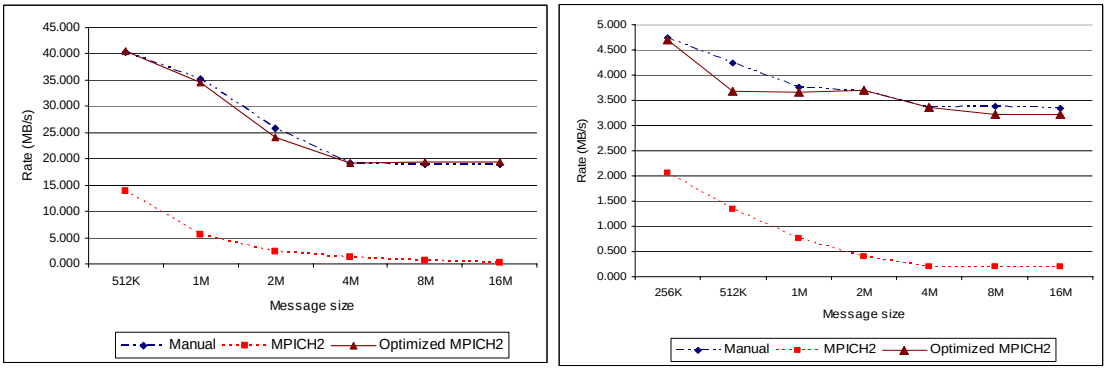


Figure 5.14. Bandwidth measurements for vector (left) and indexed (right) on sunwulf

Figure 5.15 shows the performance of programs communicating messages formed using nested derived datatypes representing a 3D-cube on the *jazz* cluster and Figure 5.16 shows that on the *sunwulf* cluster. On both clusters, the original MPICH2 performs similar to manual and optimized implementations for smaller data sizes. As the message size (size of 3D cube) becomes larger compared to the L2 cache size, the performance degrades for MPICH2, whereas the optimized implementation maintains superior performance similar to that of the optimized manual program.

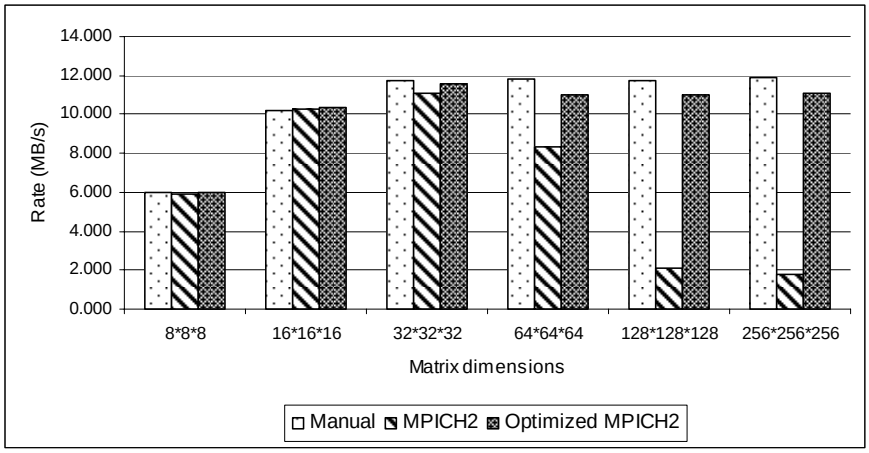


Figure 5.15. Bandwidth measurements for the 3D-cube experiment on jazz

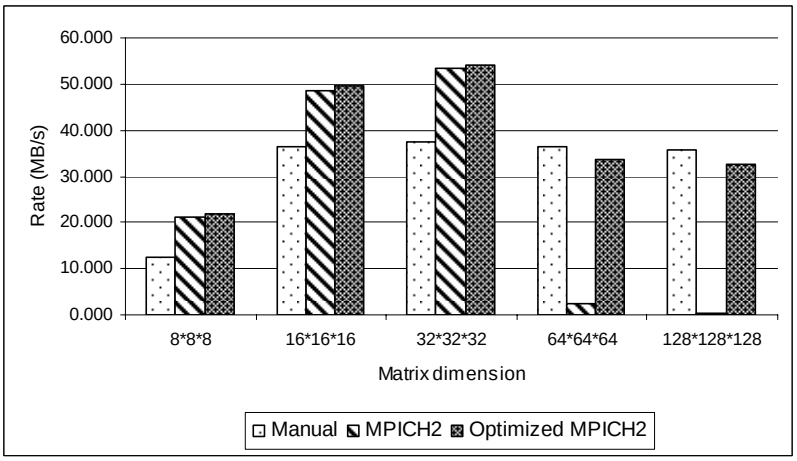


Figure 5.16. Bandwidth measurements for the 3D-cube experiment on sunwulf

Figure 5.17 shows the performance of the NAS MG benchmark on *jazz* and *sunwulf* clusters. We measured the execution time of the MG benchmark by using 4, 8 and 16 processors with B and C class workloads. The execution time with MPICH2 is higher than that of the original MG benchmark implementation (manual). With optimized MPICH2, the execution time is up to 8% (on average 6%) lower than that of manual implementation, and up to 25% (on average 13%) lower than that of unmodified MPICH2 on the *jazz* cluster. On the *sunwulf* cluster, for 8 and 16 processors, the

execution time is up to 12% (on average 7.3%) less than that of the manual implementation. Here, manual implementation is original NAS MG benchmark, which is not optimized for cache blocking and prefetching. Our optimized MPI derived datatype implementation benefits from using cache blocking in the nested datatypes in the MG benchmark.

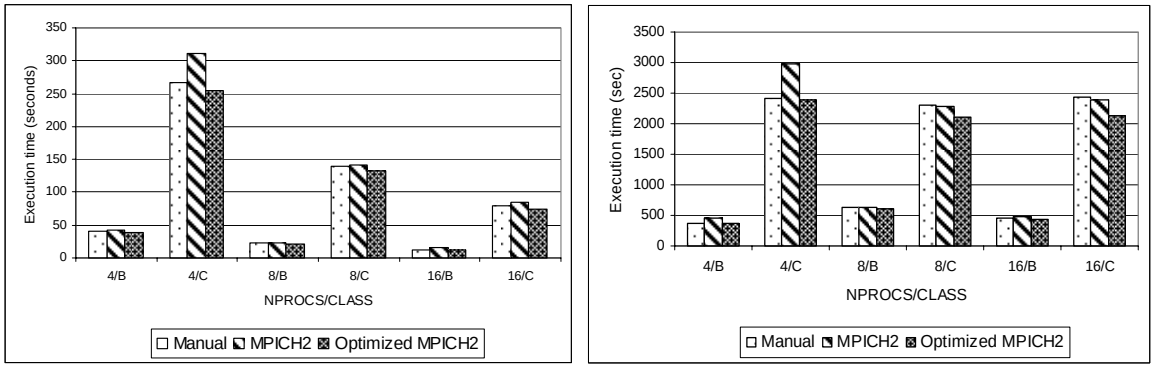


Figure 5.17. Execution time of the NAS MG benchmark on jazz (left) and on sunwulf (right)

Figures 5.18 and 5.19 show the performance (rate in MB/s) of the matrix transpose subroutine of NAS FT benchmark on *jazz* and *sunwulf* clusters, respectively. When the message size is larger than the L2 cache size, the rate degrades severely for unmodified MPICH2 because of the large number of cache misses caused by poor data locality. The optimized MPICH2 implementation benefits from using cache blocking in this program. The performance gain is in the range of 50–60% on *jazz* cluster and 50–114% on the *sunwulf* cluster.

6.5 SUMMARY

It was believed that data allocation is not a noticeable factor of communication in a parallel computing environment. All the existing parallel programming models consider cost of memory access either constant or negligible. Through our experimental testing,

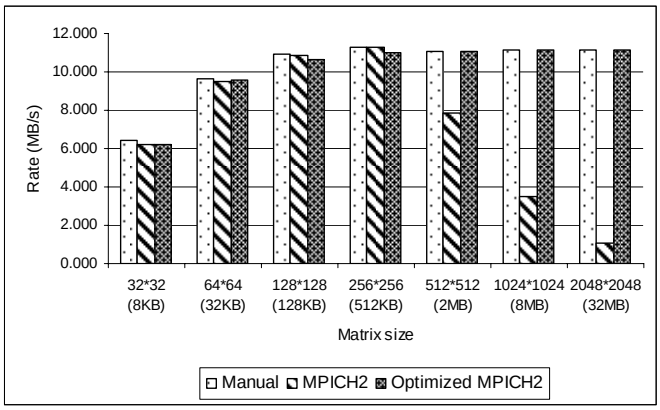


Figure 5.18. Bandwidth measurements for matrix transpose experiment on jazz

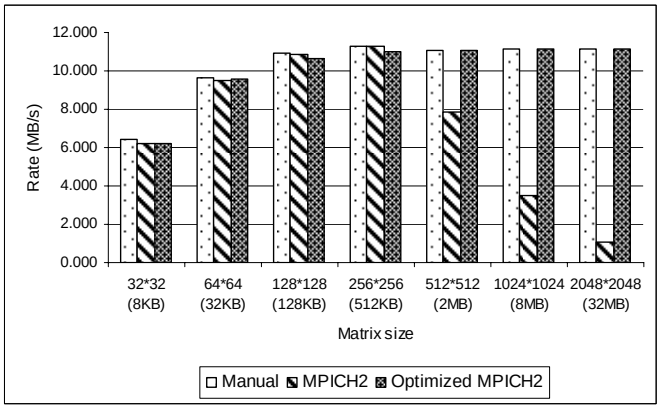


Figure 5.19. Bandwidth measurements for matrix transpose experiment on sunwulf

and case studies, in this research we have shown that memory communication is a function of data size and distribution. The performance degrades by a factor 10 times even with a small stride of 16. Communication performance can be improved more than 115% by using memory friendly optimizations external to compilers. This portrays a large scope for improvement of communication dominant applications and compilers.

Memory communication can be caused by many factors, under utilization fast CPUs with multiple levels of memory hierarchy, data distribution and various copying overheads between buffers. Application developers need to be aware of underlying architectures to develop high performance programs. But lack of documentation

regarding the memory hierarchy and buffering schemes for various architectures is a source of difficulty in optimizing applications. Identification of these implementation based communication overheads is a part of enumerating the memory communication costs. In this chapter, we have presented an approach to determine the critical data path scientifically along with the memory access overhead as described in this chapter.

By combining optimization methods with a memory access model, we have introduced in this chapter an approach to optimize memory performance automatically. The optimized implementation of MPI derived datatypes chooses packing templates that are optimized for advanced hierarchical memory systems of modern machines. These templates are parameterized with various architecture-specific parameters (for example, block size and TLB size), which are determined separately for different systems. By using these optimized templates, we obtained significantly higher performance than the existing MPICH2 implementation and manual packing/unpacking by the user. This result is significant because it will improve the performance of MPI_Pack/Unpack and MPI communication functions in many applications that use MPI derived datatypes in performing noncontiguous communication. We have shown that our optimized implementations are applicable on multiple architectures (Intel and Sun).

The optimizations described in this chapter are not yet incorporated into the MPICH2 release, but we plan to do so. We are also looking at other applications of automatically selecting optimization parameters using the analytical prediction model. For example, in scientific applications, major portion of their run time is spent in executing loops.

CHAPTER 6

PERFORMANCE RESULTS WITH DPS ARCHITECTURE

In Chapter 3, we presented the design of DPS and discussed the issues of predicting data accesses and pushing predicted data closer to a processing core. In this Chapter, we present performance results of SPEC CPU2000 benchmarks [Buab96]. As the micro-acrchitectural modifications proposed to support DPS system are not readily available in existing processors, we use the SimpleScalar toolset to simulate our DPS architecture. We first explain the structure of SimpleScalar, and implementation of DPS on this simulator, then present the simulation results.

6.1 SIMPLESCALAR SIMULATOR

To model existing complex processors, SimpleScalar toolset [Buab96] was developed. This system software infrastructure is commonly used to build modeling applications for program performance analysis and microarchitectural modeling. The toolset is derived from the MIPS-IV instruction set architecture [Pric95]. It contains various simulators, including sim-cache, sim-fast, sim-safe and sim-outorder. Out of these simulators, sim-outorder is the most detailed simulator, which supports out-of-order issue and execution based on Register Updated Unit (RUU) [Sohi90]. These simulators accept various arguments related to memory subsystem described in Chapter 3, such as cache size, cache lin e size, cache associativity, load store table size, reorder buffer size, TLB information etc.

The pipeline for *sim-outorder* simulator is shown in Figure 6.1. In the **fetch** phase, instructions are fetched from memory based on program counter and branch prediction

decisions. The **dispatch phase** models decoding of instructions into their opcodes, while also updating dependency information between instructions. In the **scheduler and execution phases,** all load instructions having their input dependencies met are first queued into the ready queue. Ready instructions from the ready queue are issued to functional units that are available. In this phase, load operations are ready with their effective addresses. The **writeback & commit stages** identify instructions that are dependent on recently completed instructions, and schedule them. They also commit the results of write operations to the cache, and retire completed instructions.

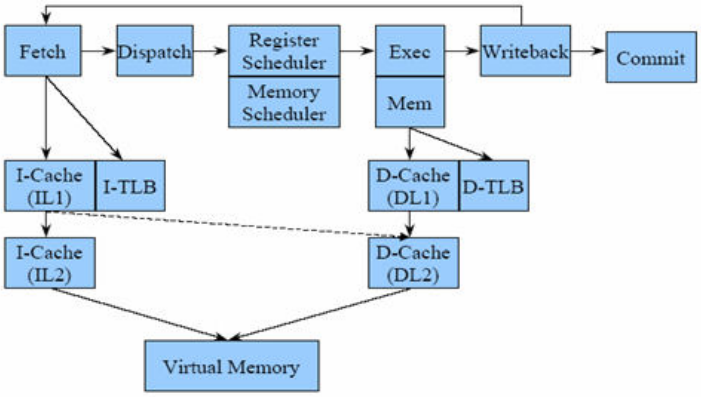


Figure 6.1. Pipeline for sim-outorder simulator

6.2 DPS IMPLEMENTATION ON SIMPLESCALAR

We evaluate the performance of DPS using an extended version of the *sim-outorder* simulator of SimpleScalar toolset V4.0. The baseline simulator configuration consists of a four-issue dynamic superscalar cores similar to that of Alpha 21264, with configuration shown in Table 6.1. Memory subsystem contains separate data and instruction L1 caches of 32 KB that are 2-way set assosiative. L2 cache memory is unified for instruction and data, which has 1 MB capacity and 4-way set associative. The size of RUU is 256.

Table 6.1. Simulator configuration

Issue width	4
Load store queue	64 entries
RUU size	256 entries
L1 D-cache	32KB, 2-way set associative, 64 byte line, 2 cycle hit time
L1 I-cache	32KB, 2-way set associative, 64 byte line, 1 cycle hit time
L2 Unified-cache	1MB, 4-way set associative, 64 byte line, 12 cycle hit time
Memory latency	120 cycles
DAH size	512 entries
Prefetch queue	512 entries

We modified the *sim-outorder* simulator to accommodate strided prefetching and DPS prefetching strategies. To apply strided prefetching, we modified the simulator using a 512 entry reference prediction table (RPT) and prefetch instructions are triggered when a cache miss occurs [Chba95]. The prefetch distance is constant and set as 8 for strided prefetching.

To simulate the DPS system, we modified the simulator to add another Alpha 21264 core that contains all the components of DPS core. Originally, the simulator only supports one processing core. We added the functionality of another core to support DPS on a dual-core processor. Operation of the second core does not affect the cycles or instructions of the processing core. To simulate data prefetching functionality, we first modified the memory module of the DPS core to introduce an instruction (`PUSH CORE_ID` as explained in Chapter 3) to push data into the prefetch cache of processing core. This instruction implemented by adding a bus to support pushing data into the prefetch cache. We also added a bus to push directly into the L1 cache of the processing core in order to verify the amount of cache pollution caused by aggressive DPS

prefetching. We modified the cache manager of processing core to support data push by the DPS core. We added a prefetch engine module to the main components (performance core) of SimpleScalar simulator to observe data access patterns and to predict future references. This includes the pattern detection manager (PDM), a DAH table, and a prefetch queue similar to the ready queue structure of the *sim-outorder* simulator. The PDM collects data into DAH. The prefetch strategy selector adaptively chooses prediction algorithms among simple strided, Markov chains, distance prefetching, and our multi-level difference table (MLDT) strategies based on the pattern information provided by the PDM. The DPS core triggers a prefetch when there are prefetch requests in the prefetch queue.

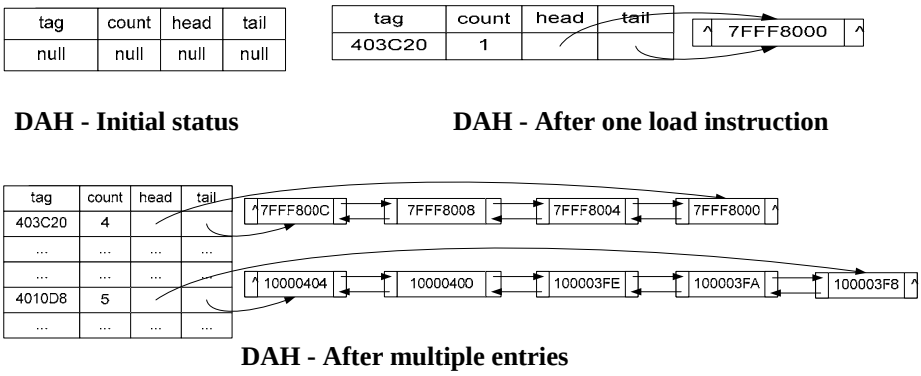


Figure 6.2. Data Access History (DAH) table

To store the history of data accesses in DPS prefetching strategy, we use a Data Access History (DAH) structure to collect load instruction information. The DAH is similar to RPT, but stores more information. DAH has a tag, count, tail and head pointer fields (Figure 6.2). Tag field records the instruction address. Each entry is a doubly linked list, which is a queue and keeps track of data access addresses and the time of occurrence (in cycles) of the corresponding entry instruction. Count field maintains the number of times an instruction with address stored in tag field has been issued. The stride

between the addresses and times are calculated to find a pattern. Figure 6.2 illustrates a constant stride is detected for instruction address 403C20 and a variable stride (a pattern with depth 2) is detected for instruction address 4010D8. This design makes DAH capable of capturing more history of recent accesses instead of only two latest accesses as in RPT, thus makes it possible to capture multi-level difference table of length *n*. Similarly, for a certain data address stored in tag field, the DAH structure manages instruction addresses to find which instruction is using data lines. In this way, DAH structure supports Markov chain pattern prediction method also, where a certain data line is accessed repeatedly.

6.3 SIMULATION ENVIRONMENT

As a first assessment of the potential of server-based data push model, we constructed a set of simple and complex regular strided pattern benchmarks. For these benchmarks, we analyzed the cache performance. We then verified the performance improvement of SPEC CPU2000 [Spec00] benchmarks that have poor L1 cache performance to test the performance gains using DPS. Table 6.2 lists microbenchmark kernels that are crucial components of well-known benchmarks such as BLAS (Basic Linear Algebra Subroutines) [Dcmd90], STREAM [Mcca95], and SPEC CPU2000 benchmarks. 2D-matrix transpose and 2D-matrix multiplication are important matrix operations in scientific applications. These two operations exist in many benchmarks that test the performance of computer architectures including BLAS, CPU2000 benchmarks. Struct kernel is taken from CPU2000 benchmarks. Figure 6.3 shows another nested strided pattern, which accesses a 3-dimensional matrix. This pattern contains repetition of three

different strides. The first stride contains accessing two contiguous elements of 1-dimensional array. The second and third accesses contains accessing two 1-dimensional and 2-dimensional arrays, respectively, with a different strides.

We select these benchmarks, as they represent data access patterns found in real

Table 6.2. Benchmark kernels

Kernel	Operation	Access Pattern

2d-matrix transpose	for (i = 0; i < N; i++)
	for (j = 0; j < N; j++)
	y[i][j] = x[j][i];
		y: contiguous
		x: non-contiguous

2d-matrix multiplication	for (i = 0; i < N; i++) {
	for (j = 0; j < N; j++) {
	t = 0;
	for (k = 0; k < N; k++){
	t += a[i][k]*b[k][j];
	}
	c[i][j] = t; }
	}
		a: contiguous
		b: non-contiguous
		c: contiguous

struct accesses	for (i = 0; i < N; i++) {
	type_a[i]->longval1 = a[i];
	type_a[i]->longval4=b[i];
	type_a[i]->longval8=c[i];
	}
		type_a: non-contiguous, irregular stride of repeating 1,64 and 64,
		a, *b*, *c*: contiguous

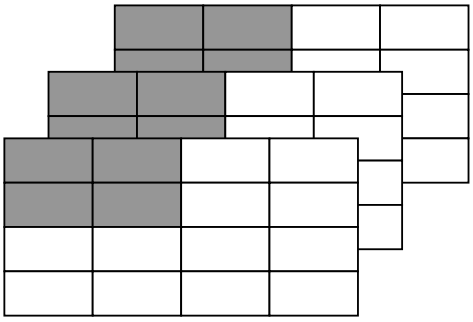


Figure 6.3. 3-dimensional nested strided data access
Innermost stride to access 1-D array, second strided pattern to access 2-D array, and the outermost strided pattern to access 3-d array.

codes and to explain how our prediction algorithms works. For instance, Matrix transpose and multiplication operations are classic examples of noncontiguous (strided) accesses that contribute to high cache miss rates when the data size exceeds the cache size. These algorithms have been the targets of numerous cache performance improvement studies. A *struct* is a user-defined datatype in C language. These *struct* accesses represent variable strided data accesses, when they are defined with different basic data types or with other user defined data structures. These accesses increase the cache misses when the stride between successive accesses is larger than a cache line.

We compare the L1 and L2 cache miss rates of all benchmarks for three cases: without prefetching (base case), with strided prefetching, and with DPS prefetching strategy. In the base case (without prefetching), the cache misses include compulsory, capacity and conflict misses. The strided prefetching strategy predicts the next stride based on the history of recent accesses and a prefetch instruction is issued on the occurrence of a cache miss. These programs were compiled with `gcc V3.2.3` with optimizations turned off to verify the effect of prefetching. We use the SimPoint [Shpc01] toolset to select a representative starting point beyond a program's initialization phase.

We also evaluate the performance of SPEC CPU2000 benchmarks. We selected five benchmarks that have high L1 cache miss rate. These programs were compiled with `gcc V3.2.3` using "`–O3 –static`". Each program is simulated for 200 million instructions after fast forwarding past the initialization phase selected by the SimPoint toolset.

6.4 SIMULATION RESULTS

Figure 6.4 shows the L1 cache miss rates of the benchmark kernels. We set N = 1024 in all the benchmark kernels, where each loop iterates for N times. 2-D matrix transpose has one contiguous access pattern (array *y*), and one non-contiguous access pattern (array *x*). When data size is more than cache size, each cache line loaded into the cache while accessing a matrix column is flushed (with row-major order) before it is reused in accessing the next column. The cache miss rate without prefetching is 56.25%. Using strided prefetching, the cache miss rate is reduced up to 26%. In accessing array *x*, there are two types of strides: forward (positive) strides to access each element of a column of the array, and a negative stride after transposing a column fully. The request generator adjusts timing to prefetch (i.e. prefetch distance is selected dynamically) with DPS strategy, where miss rate is reduced to 0.01%. These misses occur during the data access pattern learning stage. L1 cache miss rate of 2-D matrix multiplication without prefetching is high due to the number of noncontiguous accesses to array *b* (~50%). Here, arrays *a* and *c* are accessed contiguously, but strides are different. Array *a* is accessed in

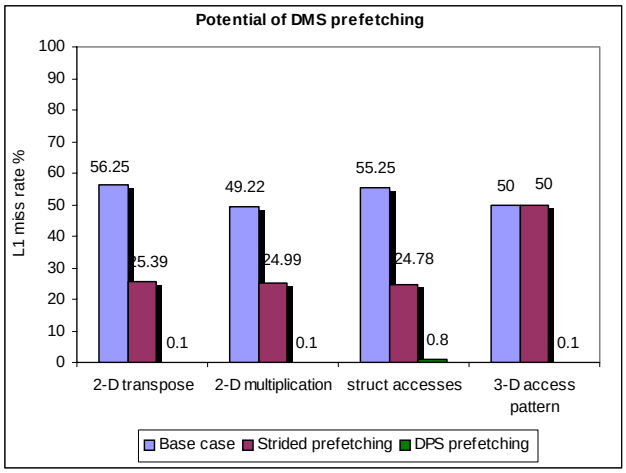


Figure 6.4. Performance of Kernel benchmarks

every iteration, while array *c* is accessed once every N iterations. Each of these arrays has reuse among the fetched cache lines, but array *b* has no cache reuse, when data size is more than cache size. The strided prefetching strategy reduces the miss rate to 24%. For struct accesses benchmark, the strides are set to 1, 64 and 64 and these strides repeat. The L1 cache miss rate with base case is 55%. In this case, DPS prefetch strategy is able to predict repeated sets of patterns. The request generator adjusts the value of *k*, as there is no reuse in accessing *type_a* structure. With this strategy, most of the cache misses are overlapped. Similarly, for 3-D access pattern, DPS prefetching is able to predict repeated sets of patterns.

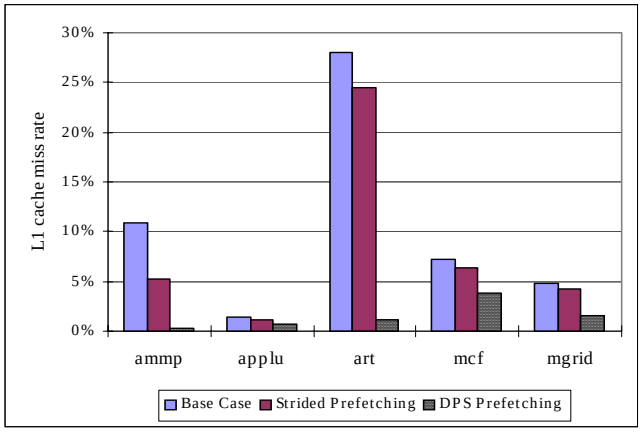


Figure 6.5. L1 miss rate for SPEC 2000 benchmarks

Figure 6.5 shows L1 cache miss rates of CPU2000 benchmarks. With DPS prefetching, L1 miss rates are reduced significantly for all the benchmarks. For *ammp* L1 miss rate reduction is 97.05%. For *applu* it is 48.9%, for *art* it is 96%, for *mcf* it is 32%, and for *mgrid* benchmark it is 66.5%. These miss rates are 40% to 95% less (66% on average) compared to strided prefetching. Figure 6.6 compares the number of L2 misses for these benchmarks, which shows significantly reduced number of misses with DPS prefetching.

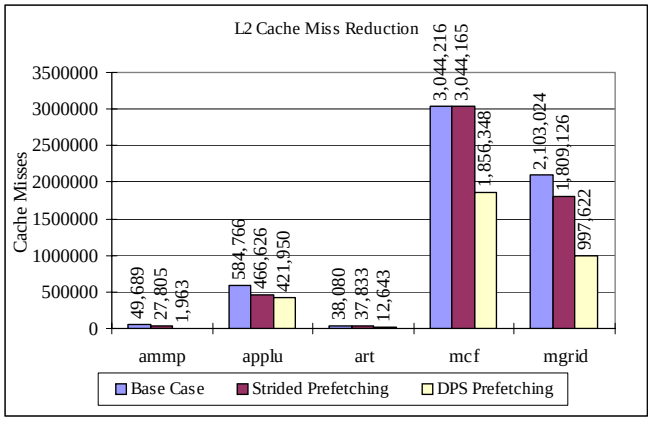


Figure 6.6. L2 misses for SPEC 2000 benchmarks

Figure 6.7 shows the values of IPC (instructions per cycle) improvement for the above CPU2000 benchmarks. The first bar shows the IPC improvement with strided prefetching. The second bar represents the IPC improvement when we implement DPS prefetching without a dedicated DPS core, i.e. DPS prefetching is implemented on the same processing core, where benchmark code is running. The third bar represents the IPC improvement, when we use a dedicated DPS core for our prefetching strategy. Strided prefetching improves IPC slightly, but degrades for *applu* benchmark. When DPS is implemented on the same processing core, the IPC improvement is negative for all benchmarks except for *ammp* benchmark. This shows that, even though aggressive DPS prefetching is effective, when it is implemented on the same processing core, the overall performance degrades. With the use of a dedicated memory server core, the IPC values improve significantly, benefiting from aggressive prefetching.

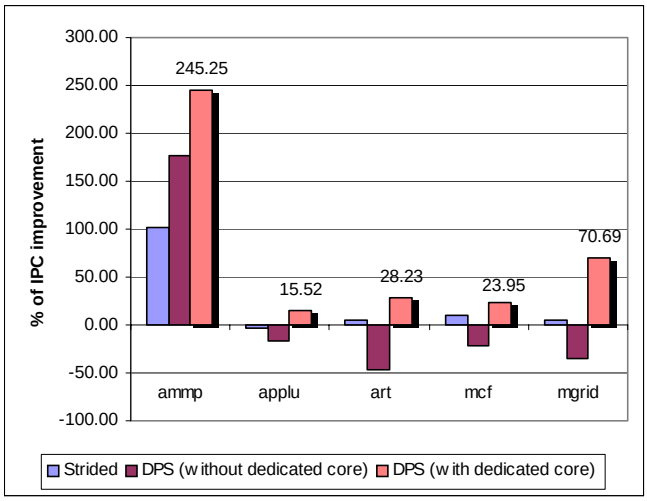


Figure 6.7. IPC improvement with DPS prefetching

Cache pollution is a negative effect of aggressive prefetching if prefetche cache lines are directly pushed into L1 cache. The cache replacement rate reflects cache pollution and it increases if data is not pushed in time. We have tested these effects as well with DPS pushing data into L1 cache instead of prefetch cache. Figure 6.8 shows the replacement rates for the above CPU2000 benchmarks. It can be observed that the cache pollution effect is none for all the benchmarks with DPS prefetching strategy. For strided prefetching, the data is prefetched only when there are regular strided patterns among data accesses. This does not increase cache pollution and cache performance improvement is also low. With aggressive DPS prefetching, the cache performance improvement is higher while keeping cache pollution low.

These performance results show the greater performance gains than existing approach by using a dedicated DPS for prefetching. We have shown that adaptability of prediction algorithms works positively towards data access performance gains. The goal of separating data access from computing is also feasible. The use of a DPS core reduces the actual prefetching overhead at processing cores and the performance gain would

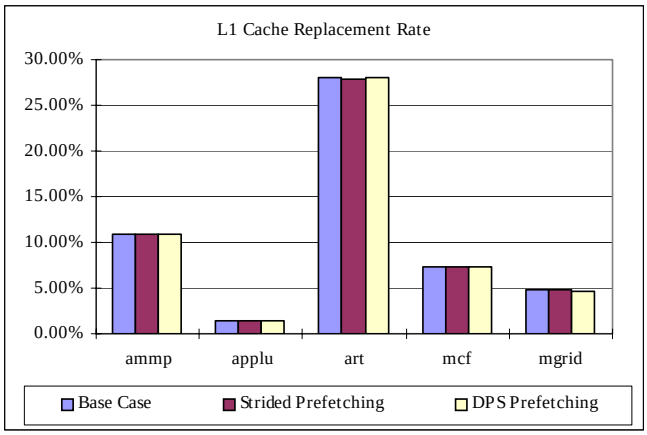


Figure 6.8. Effect on replacement rate with DPS prefetching

supercede the overhead involved in observing the patterns. Moreover, DPS has flexibility to predict temporal data access patterns adaptively, to prefetch data *in time* and to serve multiple clients. We have also shown aggressiveness of pushing data towards processing cores has less cache pollution effect, which shows the accuracy of spatial and temporal pattern prediction is high with our DPS system. These functionalities of DPS broadens the impact of CMP architectures in bridging the divergence gap of HEC.

6.5 SUMMARY

In this Chapter, we have demonstrated that DPS can improve data access performance significantly. The simulation results show that DPS has considerable gains in memory access performance for various data access patterns. DPS has reduced L1 cache miss rates of benchmark kernels with various strided patterns, in particular those of SPEC CPU2000 benchmarks to less than 1%. This is a significant improvement (up to 95%) over strided prefetching. These results show the potential of DPS in avoiding most of the processor stall time by moving data closer to computing in time.

DPS can be implemented on architectures, including SMP and clusters with special hardware and OS support. We plan to investigate DPS approach further for fast data access and to explore its prospective performance gains in other domains of information processing. We plan to extend this work to study detailed implementations of DPS and to design a strategy to select various pattern prediction strategies based on compiler and user-provided hints. This will improve the effectiveness of DPS in predicting irregular patterns such as data structure traversals. We intend to explore more accurate pattern prediction algorithms, such as time-series analysis models. In the next chapter, we explain the applications of extending data push server architecture for various levels of memory hierarchy to cross the memory-wall hurdle in High-End Computing.

CHAPTER 7

APPLICATIONS OF OUR DATA ACCESS MODELS

Data access cost models of push prefetching and cache performance prediction has various applications. We discussed the fundamentals of our models at cache memory level in the previous chapters. In this chapter, we discuss the extensions of these models. Push architecture can be applied at multiple levels of memory hierarchy to push data from lower level storage device to memory that is closer to processing unit. We present the designs of using push model for parallel computing. Energy consumption is another major research area to be explored, as the supercomputers are growing larger rapidly. We discuss the basics of finding balance between memory performance improvement and energy consumption.

7.1 MEMORY SERVERS

Data access latency is a critical performance bottleneck in shared memory and cluster computing environments as well. Data communication is an essential part of parallel computing. In shared memory multiprocessing, multiple processes share a large memory to communicate intermediate results. In cluster computing, processes use explicit message passing functions to transfer data among them. As we have shown in Chapter 5, memory communication dominates the cost of sharing data. This problem must be addressed to solve to achieve higher productivity in numerous clusters in HEC.

Another problem in parallel computing is that numerous applications require more memory than they have locally. Each node of distributed shared memory architecture or clusters has a small local memory. In DSM, the nodes share a large memory. The data

transfer activity between the local memories and disk increase, when applications require more memory space. One solution that was widely researched is to use memory servers. Many researchers have explored the concept of memory servers to reduce disk accesses and to directly use the memory of a peer node or the memory on a server, connected through network. It is considered that large collective memory capacity of a cluster of workstations is often idle, and it would be more cost effective to exploit the unused memory of these nodes. The critical assumption in this strategy is that a page access from the disk is slower than accessing a page from the global memory of a peer node or a global shared memory. This assumption is true as the network technology has made strides of improvement in reducing latency. Disk access latency is in the order of milliseconds while network latency is in the order of a few hundred nanoseconds. The trend is continuing in reducing network latency further towards a few nanoseconds.

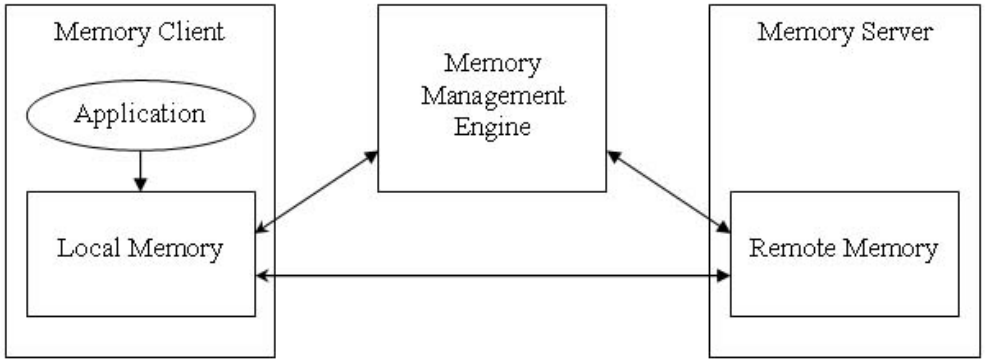


Figure 7.1. Architecture of Memory Servers

Some of the major projects with the idea of exploiting memory space over network in various computing environments are remote memory model [Cogr90] with dedicated machines to provide memory space, remote memory model with idle machines [Feza91], Global Memory Service (GMS [Fmpk95] and Network Memory Sever (NMS) [Star00].

Dodo [Acks99] suggests harvesting idle memory space by using resource monitors, a central memory manager and scheduler. Iftode et al. [Iflp93] suggests using part of the nodes of a multi-computer as memory servers. These memory servers form a layer between the RAM and disks. Recently, Li Xiao et al. [Olxl04] have proposed Parallel Network RAM (PNR) to utilize global cluster memory when memory requirement is large. Each of these projects commonly has clients that require extra memory, servers that provide memory space, and a memory management engine (MME) to help the client locate the memory server and to move the data from the server to the clients (Figure 7.1). The server is capable of providing remote memory space for multiple clients. For the ease of description, we show one client in Figure 7.1. These projects differ in defining the functionality of the MME and its position in multicomputer environment.

The goal of existing memory server research projects discussed above is to avoid slow disk accesses. However, this extra memory space provision itself is not enough in solving data access performance bottleneck problem. We plan to use our server-based data push architecture at this level. We can place the Data Push Server at MME level for parallel computing (DPS-P). The primary goal of DPS-P is to improve the data access performance of CPU by adaptively and proactively pushing the data closer to the processor. The vital functionality of DPS-P is to predict future data accesses, their access times and, to push the corresponding data to the processor *in time*. DPS-P is also responsible for managing data fetching and adaptive replacement strategies that provide better performance for various data access patterns [Suby05].

7.1.1 CLASSIFICATION OF MEMORY SERVERS

Using a DPS-P memory server that provides smart memory management is beneficial to many multiprocessor platforms. We consider two platforms that can benefit from such a server. They are: clusters and shared memory parallel processing machines. In clusters, instead of providing large memory for all nodes, it is cost effective to use a server that supports many nodes. In shared memory parallel processing environment, this server supports aggressive data fetching and prefetching closer to applications of several nodes from the shared memory. The DPS-P approach can be used here to fetch the data from the remote memory to the processes effectively to improve the performance.

A compute cluster is loosely coupled and each node has its own local memory. DPS-P can fetch data into a client at two levels; to the local memory of a client or directly to the deepest level of cache of a client. Based on the functionality of DPS-P, we classify the memory server model into two: *pure server model* and *hybrid server model*. With the *pure server model*, DPS-P manages the local memory of clients as well as the remote memory on the server and the memory provided by idle nodes. Data that is located outside the local memory of a client is pushed by the DPS-P directly into a *designated prefetch cache*. The *designated prefetch cache* is either a specially designed *prefetch cache*, or any one of the levels of cache hierarchy (L1 cache or L2 cache). If a client has no local memory, memory server automatically becomes a pure server model. In a *hybrid server model*, the memory server pushes the data to local memory of a client. The clients use their own fetching strategies to move the data to their cache hierarchy. Both of these local memory management and DPS-P memory management co-exist and need conformity to move the data in and out of the memory at the client and the memory at the

server. This hybrid model is similar to the memory server model discussed in figure 7.1, but the DPS-P provides push-based data prefetching.

In the shared memory (SMP) and multicore environments, nodes or processor cores share the memory. These environments are natural for the *pure server model*. For SMP, one of the nodes can function as the DPS-P to fetch and prefetch the data into the other nodes. This is similar to the DPS we presented in Chapter 3 for multicore processors, but has more functionality of managing memory operations.

In the following subsection, we discuss designs of DPS-P according to the classification. The fetching and prefetching algorithms used in DPS-P are similar for all the multiprocessor environments. The designs differ based on the context of the clients they are serving as well as the source and destination of the data DPS-P is moving from and to, respectively.

7.1.2 ARCHITECTURE AND DESIGN ISSUES

With the objectives of decoupling the data access from computation, aggressive prefetching, cost effectiveness, and adaptive memory management to improve the overall memory access performance, we propose a design for data push server for MME for parallel computing (DPS-P). The primary function of DPS-P is to push (or pre-push) data closer to the processor to overlap the data access latency. DPS-P also extends the virtual memory of client nodes based on the memory requirement of the applications that are running on those clients. We first discuss designs of pure and hybrid DPS-P for cluster environment and then present the designs for SMP.

There are three main components in the design of DPS-P, which are common to all three multi-processor environments mentioned above. They are prefetch engine (PFE), memory management engine (MME), and data propeller (DPR) (Figure 7.2).

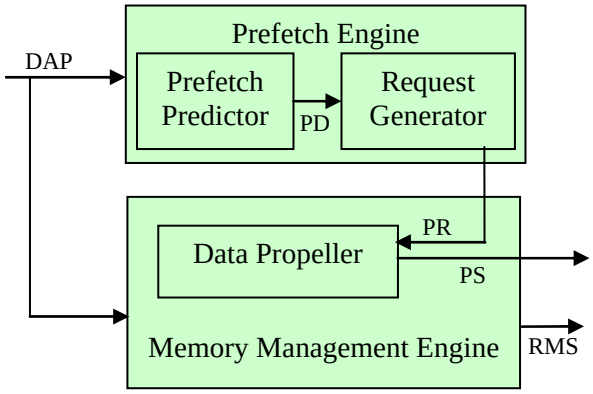


Figure 7.2. Prefetching Engine
(DAP: Data Access Pattern, PD: Prefetch Decision, PR: Prefetch Request,
PS: Prefetch Signal; RMS: Raw Miss Signal

A prefetch engine (PFE) observes the data access pattern (DAP) of an application, and predicts the future data references. The PFE contains a prefetch predictor and a request generator. The prefetch predictor uses various aggressive prefetching strategies to predict the addresses of future references based on the data access pattern. These addresses are sent to the request generator to choose the order of prefetching. The request generator predicts the time when prefetch requests should be issued and sends this information to the data propeller. The challenge of a request generator is to decide the prefetch distance in such a way that the data is pushed to the destination "*just in time*", so that the prefetched data does not cause pollution at the destination and not arrive too late. Prefetching distance is the number of references from when the prefetch is actually issued until its first use. Based on the type of data references, there are two types of prefetch engines; *cache prefetch engine* and *memory prefetch engine*. The cache prefetch engine observes the patterns of past cache line references and predicts the future cache line

references. The memory prefetch engine predicts the memory page references by observing the page access pattern of the application. The data access pattern prediction phase can be bypassed with the use of compiler-generated hints or application provided data access pattern, if available.

The data propeller (DPR) maintains the information of the location of the data (memory at DPS-P, remote memory, or local memory) and issues prefetch instructions to the appropriate location. These prefetch requests are forwarded to the memory management engine (MME). The function of the MME is similar to that of the memory manager in the current operating systems. The difference is that the MME is located on a server, remotely. In addition, the MME decides an effective way of transferring the data from the data location to the destination. It is also responsible for fetching the data to the client's memory when there is a cache miss or page fault (raw misses) based on the function of the memory server. The MME chooses the most effective replacement policy among a pool of replacement policies, which is appropriate for the current data access pattern of the client's application. This increases the adaptivity of DPS-P to tune the workload of each application.

The placement of the three components and, their input and output differ for clusters, SMP environment. In the *pure server model for clusters*, DPS-P observes the cache access pattern and predicts the future references to push the data closer to the application. The architecture is shown in Figure 7.3. In this figure, Compute Node 1 and Compute Node 2 are the clients for DPS-P. The server node has a large memory and disk to support the applications running on the client nodes. Compute Node K is providing its idle memory to be used by DPS-P, when node K is idle. We assume that each client node

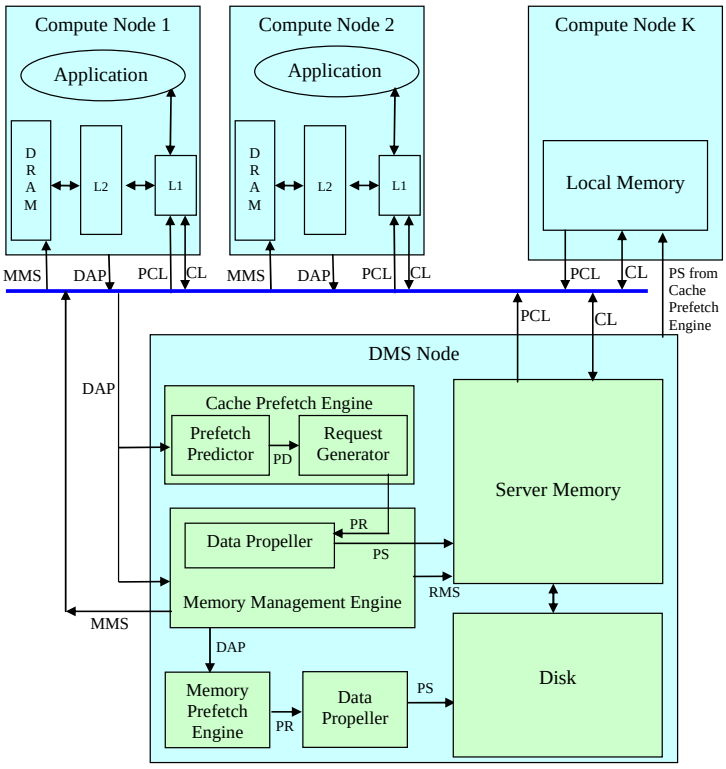


Figure 7.3. Pure Memory Server Model for Clusters
(DAP: Data Access Pattern, PD: Prefetch Decision, PR: Prefetch Request, PS: Prefetch Signal, CL: Cache Line, PCL: Prefetched Cache Line, RMS: Raw Miss Signal, MMS: Memory Management Signal)

has L1 and L2 level cache and local DRAM. The memory management engine (MME) of the server maintains the server memory as well as local DRAM of the client. The server sends the memory management signals (MMS) to the client's local memory. DPS-P uses two levels of prefetching engines, one to prefetch the data from DPS-P to the client, and another to prefetch the data from the disk to the memory at the server node. In this figure, we show that L1 cache as the *designated prefetch cache*. We assume that the server has access to the bus to push the cache lines directly to L1 cache.

In *hybrid model for clusters*, the OS of the client itself maintain (Figure 7.4) the local memory of a client. The role of DPS-P is to observe the data access pattern, to predict the future page accesses, and to push the predicted data into the local memory of the client.

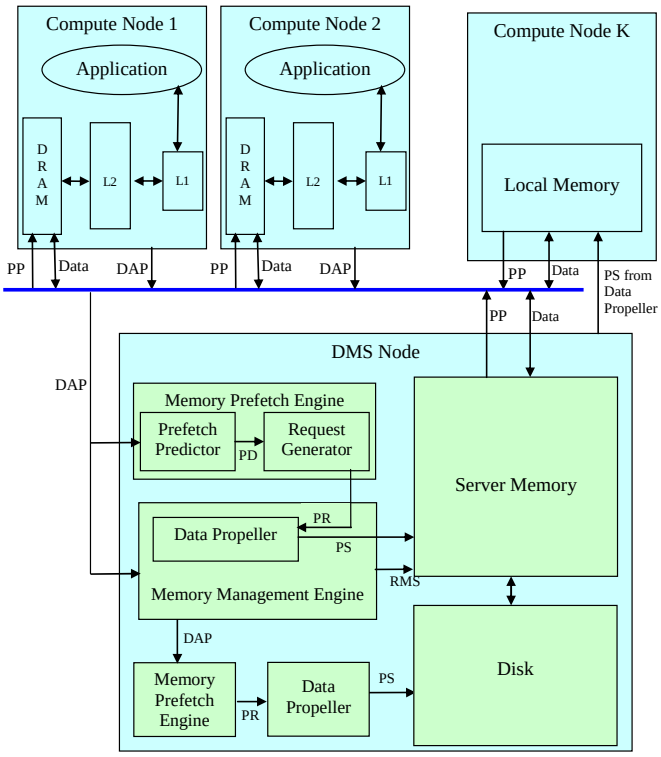


Figure 7.4. Hybrid Memory Server Model for Clusters
(DAP: Data Access Pattern, PD: Prefetch Decision, PR: Prefetch Request,
PS: Prefetch Signal, PP: Prefetched Page, RMS: Raw Miss Signal)

This server also provides extra memory space on the server or on the memory which is idle at peer nodes. The DPS-P has a memory prefetch engine to predict the future page references and a data propeller to push the corresponding data to the local DRAM of the client node. In Figure 7.4, Compute Node 1 and Compute Node 2 are the clients for DPS-P. Compute Node K is providing its idle memory to be used as extra space by DPS-P. The memory management engine (MME) of the server maintains the server memory. This DPS-P also maintains two levels of prefetching engines similar to *pure server model*. The major difference between pure model and hybrid model is that, in hybrid model DPS-P does not issue any management signals to the local DRAM of clients.

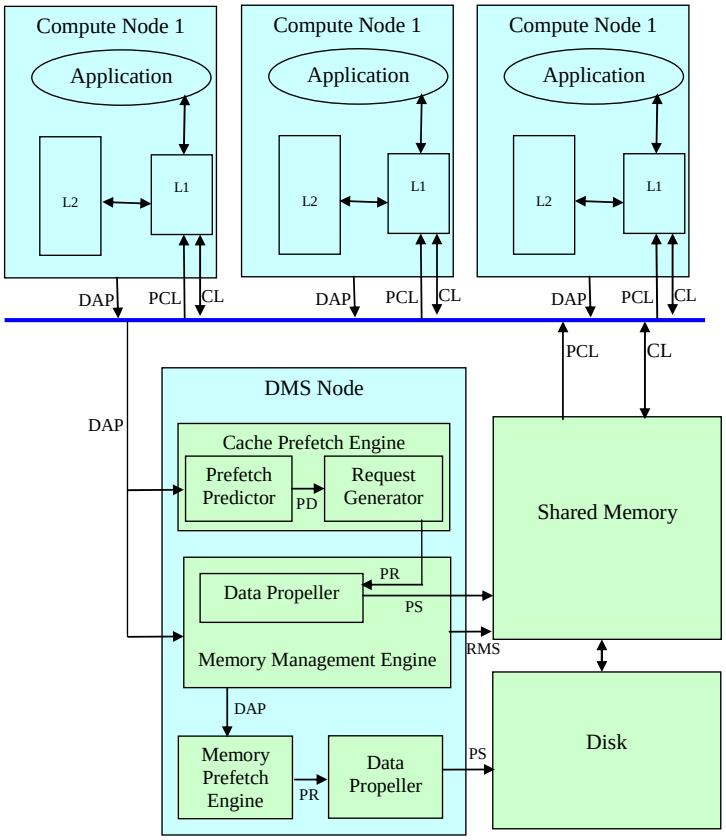


Figure 7.5. Pure Memory Server Model for SMP
(DAP: Data Access Pattern, PD: Prefetch Decision, PR: Prefetch Request, PS: Prefetch Signal, CL: Cache Line, PCL: Prefetched Cache Line, RMS: Raw Miss Signal)

In the SMP environment, clients do not have local DRAM and all the memory is shared. The DPS-P (Figure 7.5) is a *pure server model* type and it manages the shared memory. Compute Nodes 1, 2 and 3 are the clients for Memory Server Node in this figure. The DPS-P observes the access pattern of cache memory of each client and predicts the future cache accesses. Then, the server pushes the corresponding cache lines to the respective client. The design of DPS-P is similar to that of the pure memory server for cluster environments. There is a cache prefetch engine to predict the cache references of clients. The difference is in the function of MME. Here, the MME is only responsible for managing the shared memory. This design also has a memory prefetch engine and a

data propeller to push the data from the shared disk to the shared memory. Again, we assume L1 cache as the designated prefetch cache at the client nodes in Figure 7.5.

In all the proposed designs, the core of DPS-P is similar, which includes the prefetch engine, the data propeller, and the MME at multiple levels of server's memory hierarchy. These are the components that are common for memory server in any environment. They make the DPS-P aggressive, and adapt to the data access pattern of the application. The aggressiveness and adaptive behavior of our model is aimed towards improving the memory access performance and bridging the gap between the peak computing capacity and sustained performance of current and future high-end computing machines.

7.1.3 INITIAL RESULTS

For irregular data access patterns, pattern prediction cost is high. When multiple variables have different patterns, identifying the pattern for each of these variables requires storing more history of references and analysis of these references to find if there is a pattern at all. In these cases, to reduce this cost caused by pattern learning phase, DPS-P can utilize the hints from the application or the compiler. To determine the feasibility of DPS-P in these cases, we performed some experiments, where the hints are provided by an MPI application. The MPI (Message Passing Interface) Standard is widely used in parallel and distributed computing [Grla99, Grld99]. One of the important features of MPI is derived datatypes, which enable users to describe noncontiguous memory layouts compactly. When the MPI application uses these derived datatypes, the user defines the noncontiguous pattern of the derived datatypes in the form of vector, struct, and indexed datatypes. These definitions contain the strides between successive elements. DPS-P can utilize the information of access pattern from these definitions to

predict the future references when the MPI application uses these derived datatypes while packing or communicating the corresponding data.

In our performance analysis, we used the examples of derived datatypes described in Chapter 5 of “Using MPI” book by Gropp et al. [Grla99]. In these examples, the derived datatypes represent various noncontiguous patterns of sending particles in solving *N-body problem*. N-body problem is one of the most central computational problems in physics. In implementing the simulations of n-body problem on multiple processors, each processor needs the information of particles of all the other processors. Each of these processors first collects (pack) the information of various particles that are located noncontiguously in the particle array and then send that collected message to other processors. In this case, if the application gives the definition of derived datatype to DPS-P, the server can push the data from the array closer to the processor during data collection phase. The definition of derived datatype contains the number of contiguous chunks of data to be collected, the length of each contiguous chunk of data and the strides between them. Without DPS-P strategy, the cache performance during this collection phase is low when there is no reuse among the fetched cache lines. With DPS-P, the number of cache misses for this collection phase can be completely avoided by prefetching *in time*.

We assume that there is a function defined to send the pattern information from the application to the DPS-P node. The derived datatype is an indexed type that collects 1024 blocks of data. The length of each block is specified in an array, and the values are [1, 2, 4, 8, 1, 2, 4, 8,…, 1, 2, 4, 8], where block lengths repeat with 1, 2, 4 and 8 *double* data elements. The strides (in bytes) between these blocks are [8, 16, 32, 64, 8, 16, 32, 64,…,

8, 16, 32, 64]. Figure 7.6 compares the cache performance of collecting data from a particle array without prefetching and with DPS-P prefetching strategy. Cache reuse with base case is only 25% while accessing the first two references in each cycle. The remaining 75% of the accesses cause a cache miss. Using DPS-P prefetching strategy with application hints regarding the data access pattern avoids all the cache misses by pushing the corresponding data to the processor from the client. The hit rate increase is four-fold. This result shows, with some language or middleware support for user input, DPS-P can further be enhanced to reach its potential in pushing the data on time without a cold start in predicting the data access patterns.

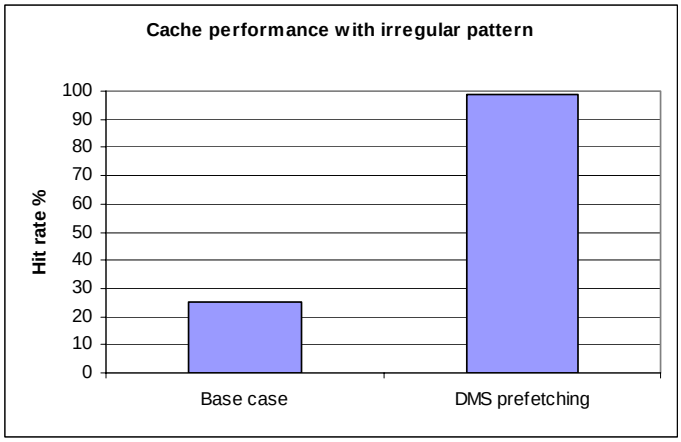


Figure 7.6. Performance improvement for irregular patterns with pattern hints

7.1.4 SERVICE ORIENTED ARCHITECTURE FOR MEMORY SERVERS

Service Oriented Architecture (SOA) is driving business infrastructure lately. The architecture of memory servers is similar to SOA. The common memory server architecture (Figure 7.1) and general process of SOA (Figure 7.7) have similarities. In both architectures, there are clients that require memory service similar to a service

consumer of SOA, and multiple servers that provide extra memory space for the consumer entities. Memory Management Engine (MME) can act as a directory service between memory clients and memory servers such that MME collects the information of available servers and advertises that information for memory clients to choose a server. Moreover, the MME can also behave as a service provider entity to predict the future data references of memory clients and push that data from its location at memory servers to the clients. This adaptation of SOA is needed as cluster computing and SMP are growing to be heterogeneous in the near future.

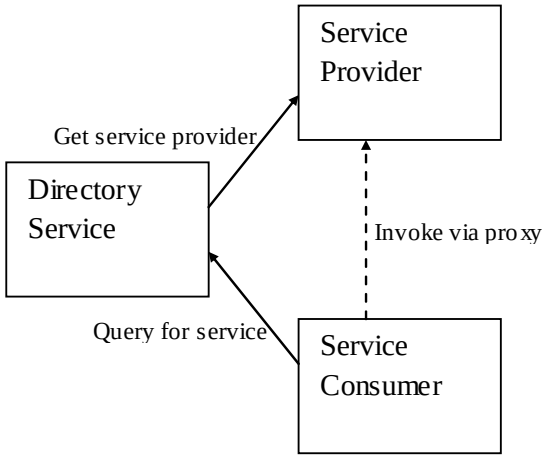


Figure 7.7. Directory service of SOA

SOA provides security, scalability, fault tolerance and interoperability. These features are beneficial for memory server architecture [Bysu06]. Among the memory server systems described in the previous subsection, some of them suffer from scalability and fault tolerance problems. Providing dedicated servers and offering memory services, such as extra memory space and proactive data movement, improves the scalability and fault tolerance.

Although SOA features are helpful for providing memory servers, the success depends on resolving several challenges. It is widely considered that the performance of

services over the web (web services) is not beneficial to high performance computing. However, SOA is not limited to web services. By providing a service-oriented infrastructure (SOI) that is modified to fit the goals of memory servers, benefits of SOA model can be applied for memory servers.

While MME provides discovery service, performance improvement depends on how often this service is needed. The interaction between memory client and MME to predict future references may affect the performance. To improve the performance effectively, it is possible to have data access profiles of clients before hand as proposed by DPS-P.

7.1.5 SUMMARY OF MEMORY SERVERS

In section 7.1, we have introduced the DPS for parallel computing (DPS-P) methodology. Hierarchical memory systems have been developed to hide the performance disparity. However, memory hierarchy is based on data localities. It works well for some applications but not well for others. DPS-P separates the data accesses from the processing unit and employs adaptive memory management and aggressive prefetching by observing the data access patterns of applications. It provides on-time data delivery services to push the desired data to the processing element. In this way we give a general solution for data access in a computing system.

The term *memory server* has been used in the past decade to provide extra memory space in reducing disk accesses. This traditional approach does not offer data access as a service, and lacks the aggressiveness and adaptability to various application workloads to move the data to the processing unit *on time*. The performance gain of these traditional memory servers is not significant. DPS-P, however, separates data from computing and pushes data to the computing element *on time*. To put the concept into practice, we have

presented different DPS-P designs to improve the memory access performance at various levels and for different multiprocessor architectures.

In a cluster environment, DPS-P can be used as a level of memory hierarchy to provide extra space for swapping pages in and out. We propose two models for this environment: pure server model and hybrid server model. With the pure server model, the local and the remote memory are managed by the DPS-P, and with the hybrid server model, the local memory is maintained by the client and the remote memory is maintained by the server. In the pure server model, data is pushed into the client's cache and the granularity of the data is a cache line. In hybrid server model, the prefetched data is pushed to the client's local memory and the granularity of the data is a memory page.

In SMP environment, the memory is shared by multiple nodes. DPS-P fits well at the memory management level in these environments. DPS-P pushes the data from the shared memory to the cache levels of the client nodes. In an SMP, one or more of the nodes could run the DPS-P operations.

Our initial results with the MPI derived datatypes show that DPS-P improves the cache hit rates of an MPI application by more than four-fold. We have shown that DPS-P can remove most of the CPU stall times by moving data closer to the computing on time, for different applications and on different platforms.

Since DPS separates the data from computing, its impact is fundamental and is beyond the field of high performance computing. For instance, it can serve as the *μ proxy* between the file server and the clients in a distributed file system to improve the scalability; can enhance coherence to provide a single image in a parallel system; and can virtualize storage in a Grid environment. Even in high performance computing, DPS can

be enhanced in language, compiler and scheduling, and can be implemented at system or application level.

7.2 HIGH END COMPUTING I/O

In previous chapters, we have demonstrated that there is a great potential to apply server-based data push architecture at memory level. This architecture can be applied at disk level to improve the performance of I/O accesses in data intensive scientific applications. In this section, we present design of File Access Server (FAS) with the goal of applying push model to I/O.

7.2.1 FILE ACCESS SERVER (FAS)

The goal of File Access Server is to "push" data pro-actively, *in time* to client's main memory. Here, "push" means the data is sent before an I/O request is generated by the client; by "*in time*", we mean that data is moved from its source to destination within a window of time before it is required, and where it does not replace other data blocks from I/O cache falsely. By moving data into an I/O cache too early, it may replace data blocks that would be accessed in the near future. Our strategy aims to avoid such negative effects. The goal of implementing this file access server is significant reduction in time-to-solution of various I/O intensive scientific and numerical applications.

In most of the scientific and numerical applications, I/O accesses show patterns ranging from simple strided accesses to complex non-contiguous patterns [Cacr95, Bawu96, Nkpe96, Smre98]. With the use of adaptive and advanced prediction algorithms to capture these patterns spatially and temporally enables our system to separate data movement from computing and to make this data available for processing by the time it is

required. Using these advanced strategies requires extra computing power at the server. However, with modern multicore chips and multiprocessor servers, we assume that the additional computing needed to analyze and predict access patterns is available on the server machine.

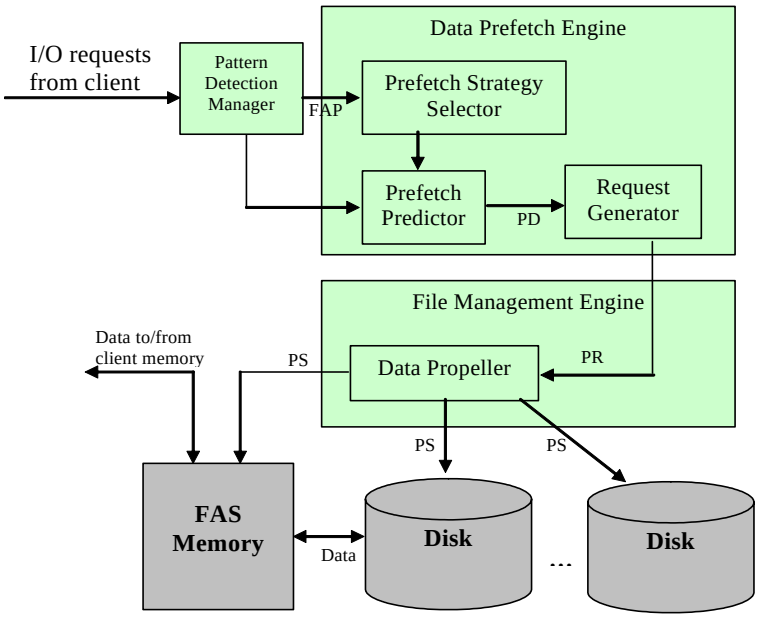


Figure 7.8. File Access Server
(FAP: File Access Pattern, PD: Prefetch Decision, PR: Prefetch Request, PS: Prefetch Signal

The structure of the File Access Server (FAS) system is shown in Figure 7.8. FAS consists of four primary components: memory, disk, data prefetch engine, and file management engine. FAS memory is the server-side disk cache. FAS conducts two-level prefetch; prefetch data from disk to the server-side disk cache and prefetch data from server-side disk cache to client-side disk cache. The data prefetch engine is responsible for decision-making. A *Pattern Detection Manager* (PDM) is introduced in FAS prefetch engine to collect the history of past I/O access patterns in spatial and temporal dimensions. A *Prefetch Strategy Selector* (PSS) adaptively selects an appropriate method

to predict future accesses based on spatial pattern information. The *prefetch predictor* of the data prefetch engine decides *what* data to fetch; the *request generator* decides *when* to fetch the data so that the prefetched data arrives *in time*. The *data propeller* of the file management engine carries on the prefetching and pushes the data into the appropriate disk cache. The data propeller is also responsible for storing the prefetched references so that it does not push duplicate data blocks. This information is also useful in designing a better replacement policy for the prefetch cache (I/O cache). If the prefetching fails, the file memory engine handles the page fault as traditional file servers.

FAS is different from traditional file servers in that it proactively pushes data to the clients, before they request. FAS is also quite different from the existing network memory server approach [Iflp93]. It does not lease its memory space. FAS's memory is its disk cache for fast data delivery to its clients. FAS has many advantages. In addition to improving data access performance, FAS provides a separation of data movement from processing. The separation gives the abstraction a user needs and makes many of the current I/O layers unnecessary. FAS collects the data access information and makes decisions of prefetching, thus the information and perspective will not be lost from one I/O layer to another I/O layer. FAS merges small noncontiguous I/O requests made from clients and pushes that data to the clients in time. FAS takes data access as a service and represents the current trend of service oriented computing. It is a natural product of the rapid advance of technologies. FAS trades data access time with extra computing power, extra memory, and fast communication. This trade off may not have been worthwhile until now. With the continually enlarged gap between CPU speed and I/O speed, the FAS design has a great potential for many years to come. This approach also provides high

scalability because multiple FASes can be set-up to function as a scalable parallel file system (see Figure 7.9). In fact, we plan to implement the FAS functionality in the file servers of the PVFS2 file system, so that this feature can be tested and used in practice.

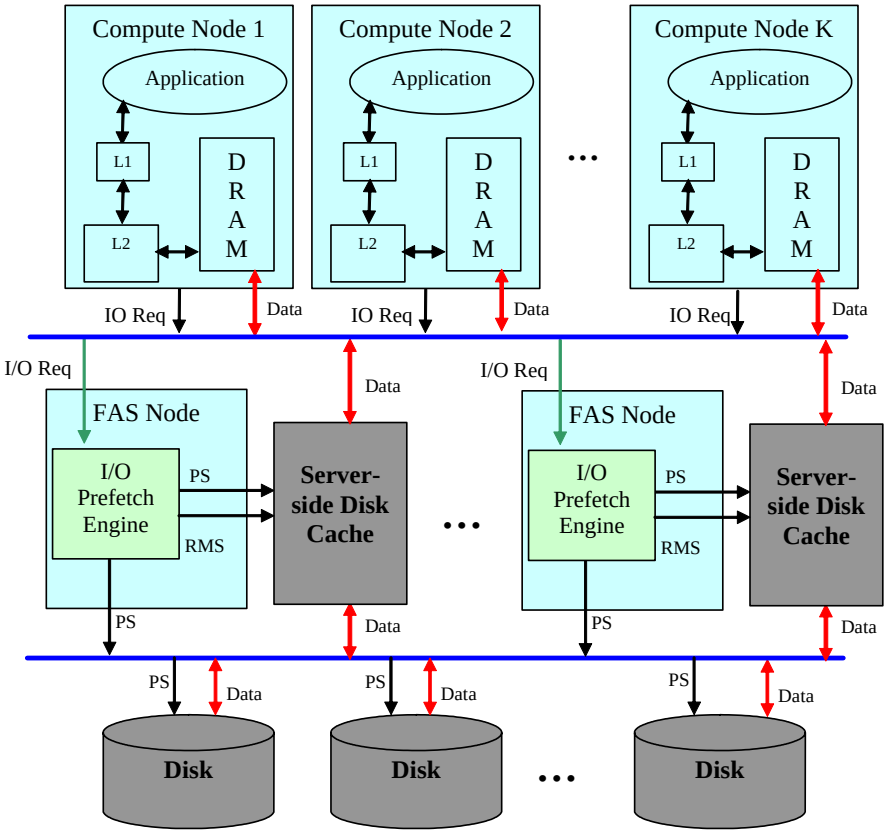


Figure 7.9. Push-based Data Movement
(PS: Prefetch Signal, RMS: Raw Miss Signal)

7.2.2 HIGH END COMPUTING I/O RELATED WORK

While the "Server-Push" model is new and has never been studied, there is a significant amount of research in optimizing I/O access performance for HPC. Research in software optimizations to improve I/O performance can be roughly classified into the areas of advanced compilers [Brms95, Bckk95, Bocr96, Modk96], runtime I/O libraries [Cffh95, Mpif96, Pggs95, Grap95, Molg92, Ledu97, Vech99, Alpb02] and parallel file

systems [Clrr00, Cfsi00, Scha02, Nase04, Trre04]. Many research groups have also proposed network memory servers to reduce disk accesses and to let clients directly access memory remotely with an assumption that data can be accessed faster on network than from disks [Fmpk95, Cogr90, Feza91, Iflp93, Acks99, Olxl04]. Many prefetching strategies were also proposed in the context of hardware data prefetching.

I/O PERFORMANCE OPTIMIZATION

Many researchers introduced advanced compiler I/O optimization techniques. Brezany et al. [Brms95] have developed a parallel I/O system called VIPIOS that can be used by an optimizing compiler. Targeting out-of-core datasets, Bordawekar et al. [Bckk95, Bocr96] have presented several algorithms to optimize communication and to reorder stencil computations. Mowry et al. [Modk96] have developed compiler inserted I/O prefetching for out-of-core applications. A common problem with compiler optimizations is that they are not effective with dynamic nature of I/O accesses. Prefetch instructions statically inserted at compile time are not adaptive to changes in network traffic when disks are located remotely. Moreover, making compilers perform complex analysis of prefetching predictions increases compilation time severely.

There has been significant amount of research effort in optimizing I/O performance using runtime libraries [Cffh95, Pggs95, Grap94, Grap95, Mpif96]. File prefetching in I/O performance optimization has been done in many approaches. Patterson *et al.* [Pggs95] have suggested modifying compilers to allow programmers to provide the operating system with hints about future file use. Kuenning *et al* [Kuen94] built SEER, a prototype file hoarding system based on the order in which files are referenced during periods of connection. Griffioen and Appleton [Grap94, Grap95] proposed an automatic

file prefetching based on the prior access stream. They maintained a probability directed graph for successors to each file and a weight is calculated for each edge of the graph based on the frequency of succeeding files. Various pattern based file prefetching methods have been proposed [Molg92, Ledu97, Vech99, Alpb02, Hire03]. These methods concentrated in improving the accuracy of predicting *what* to prefetching but failed to schedule adaptively and automatically *when* to move this data to the I/O caches. This scheduling of prefetching is necessary to maximize I/O performance.

Parallel file systems such as Lustre [Cfsi00], GPFS [Scha02], PanFS, [Nase04], PVFS [Clrr00], and PPFS2 [Trre04] are popular in enabling concurrent I/O accesses from multiple clients to files. All these file systems provide high bandwidth for large, well-formed parallel I/O requests, but perform relatively poorly on other less-ideal access patterns. PPFS2 [Trre04] offers better runtime optimization for caching, prefetching, data distribution and sharing compared to other file systems.

Despite all the effort, each of these areas of research is lacking aggressiveness in reading, writing, and moving data around fast enough to avoid severe performance bottlenecks.

PREDICTING FUTURE I/O ACCESSES

Numerous prediction methods have been proposed for hardware data prefetching and can be applied for I/O caches. These prediction methods range from very simple sequential prefetch strategies to Markov prefetching, using compiler hints in prefetching and chasing pointers. The immediate next block of data[*] is fetched with the requested block of data in One-Block-Lookahead (OBL) strategy. Dahlgren et al. [Dads93]

[*] In the context of I/O caches, a block of data is a page. A block of data for hardware data caches is a cache line.

extended this method to fetch the next *k* blocks of data to improve the spatial locality. An adaptive sequential prefetching strategy is proposed by the same authors [Dads95] to adjust the value of *k* based on the efficiency of prefetching. The drawback of either fixed or adaptive sequential prefetching is that when the stride between data accesses is large, the number of unnecessary cache blocks becomes large.

Various strategies have been proposed [Chba95, Fupa91] based on stride between successive accesses. These strategies maintain a reference prediction table (RPT) to record the recent accesses and to predict the next stride. Chen et al. [Chba95] proposed an aggressive arbitrary stride prefetching strategy. To capture the irregularity in accesses, Markov prefetching strategy [Jogr97] was proposed. This strategy assumes that history might repeat itself among the accesses and builds a state transition diagram with states denoting the accessed data block. Probability of each state transition is maintained, so that least probable predicted data references can be dropped from prefetching. Kandiraju et al. [Kasi02] have presented another strategy, which maintains state transition diagram of distances instead of references as in Markov prefetching.

While most of these prefetching strategies concentrate on *what* data to prefetch, we have found only one significant effort in predicting *when* to prefetch. ARIMA [Trre04] performs time series analysis to schedule prefetch instructions. In our push-based strategy, we plan to use all these strategies by adaptively selecting an appropriate method to predict future file references based on detected I/O access pattern.

7.2.3 TECHNICAL CHALLENGES

Challenges such as what data to push and when to push are addressed in Chapter 3 of this dissertation. We can apply similar approaches for I/O level data push. Future data

access prediction algorithms can be adaptively chosen based on history of accesses or page faults. Other challenges that are exclusive for I/O are replacement strategies and collective I/O. Replacement policies can be improved further with the use of predicted information.

When I/O cache is full and a new data block has to be prefetched into that I/O cache, it is necessary to avoid replacement of data blocks that are useful in the near future. Traditionally, LRU replacement policy is popularly used due to its simplicity of implementation. However, LRU policy does not capture “frequency” of usage of a data block. For example, if a data block is used frequently, but not accessed recently, it will become a victim block even if it will be accessed frequently in the near future. Adaptive Replacement Cache (ARC) [Memo04] replacement policy was proposed to capture the “recency” and “frequency” features of data blocks. This policy maintains two logical LRU lists. One list contains the metadata of blocks that were accessed only once “recently”, while the other list maintains the metadata of data blocks that have been accessed at least twice “recently”. The first list captures the “recency” and the second list captures the “frequency”. ARC adaptively increases the sizes of these lists based on the application workload and chooses a data block for replacement from the list that has more data blocks than I/O cache can accommodate. Even with ARC strategy, there is still a possibility of a data block that would be used in the near future is replaced.

Our File Access Server can utilize the knowledge of I/O access patterns predicted by prefetching strategies. We call this strategy Prediction-based Adaptive Replacement (PAR). In this strategy, after a victim data block is selected by ARC policy, we verify that if the selected victim block address is in the list of predicted addresses by prefetch

strategy mentioned in the previous section. If the reference of victim data block is in the list of predicted pattern, it means that data block would be accessed by the client of the I/O cache in the near future and shall not be replaced. Another data block is selected, which meets the criteria of ARC policy and verified with predicted addresses in the prefetch queue. This process continues until a data block is selected, which will not be accessed in the near future according to the prefetch engine predictions. This strategy provides an extra level of surety that a data block, which would be accessed in the near future, is not replaced.

In many parallel applications, multiple processes often need to access different portions of a file simultaneously, a pattern that is commonly known as collective I/O [Thgl02]. For example, each process may need to read or write a subarray of a multidimensional array. In MPI programs, the user often provides the file access information of a group of processes. Implementation of MPI-IO [Thgl99] uses this information to merge these requests of different processes and services the merged request through collective I/O. We plan to enhance our access pattern detection and prediction algorithms to be aware of collective I/O and perform optimizations based on the collective requests from multiple processes. In particular, we plan to use user provided hints as well to merge these small I/O requests, and push the data to the appropriate processes *in time* and optimize the performance further.

7.2.4 INITIAL RESULTS

We modified SimpleScalar simulator [Buab96] to test the performance of FAS model, similar to the modification discussed in Chapter 6. We ran simulations of file accesses with different strides using modified. In this simulation, we recorded the page-

hit rates in accessing memory for uni-processor system, while accessing a file from disk. Using prefetching strategy of DMS, page hit rates are above 95% for regular access patterns, which are far better, compared to the cases of without using prefetching and using simple strided prefetching. (See figure 7.10).

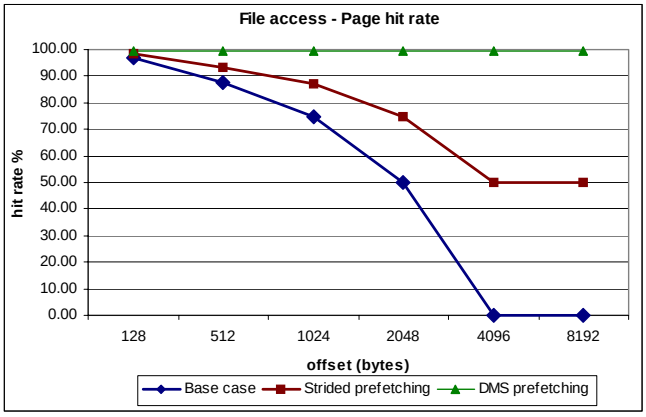


Figure 7.10. Memory performance comparison for file access kernel

The FAS approach is different from traditional client-directed prefetching and has an improved performance. Traditional prefetching requires the client either to know what to prefetch based on user-provided hints or to fetch based on predicting future access patterns. In reality, users rarely provide access pattern hints, and predicting future access patterns requires compute power at the client side that will be taken away from the user's application running on the client machine. Even the client has good hints, the information are often lost through the I/O layers. In addition, without the control of the server, the client is hardly to conduct *in time* prefetching, even we assume it has extra computing power to spend. On the other hand, server machines are more powerful, and when access-pattern analysis is performed at the server, it can be done based on the combined requests of multiple processes and can therefore better meet the needs of parallel applications. In addition, modern scalable parallel machines, such as the IBM BG/L and Cray Red Storm,

have very limited operating systems on the client; I/O system calls are simply forwarded from the clients to an intermediate I/O node, which in turn does actual file I/O. In such systems, our server-based prefetching would work even better. The Server-Push I/O architecture overlaps the processor stall time during data access more effectively and in turn reduces the execution time of application running on the memory clients. It has great potential and should be explored further.

7.3 ENERGY - PERFORMANCE TRADEOFF

Increase in the usage of portable and mobile computing and communications ignited a new topic for research: reducing power consumption. Most of the portable computing is based on battery-powered devices. Saving power whenever possible increases their usability time. This topic is expanding to high-end computing too. It is obvious since the latest processors are racing towards giga-hertz of frequencies. This also makes the HEC machines power hungry, as power consumption is proportional to the frequency.

7.3.1 POWER CONSUMPTION BASICS

In general, *electrical power* is the rate at which electrical energy is converted to another form, such as motion, heat or electromagnetic field. This is measured with a unit called Watt (W). One watt of power is resulting from energy dissipation, conversion, or storage process equivalent one Joule per second. In a DC circuit, power is the scalar quantity and is equal to the product of Voltage (V) and Current (I). *Electrical energy* is the energy made available by the flow of electric charge through a conductor. Energy is measured in Joules. Power is the metric at a discrete point of time and energy corresponds to a period of time. Energy is the product of power over a period of time.

Energy = Power * Time

$$= \int_{t_1}^{t_2} P dt \qquad \text{---} \qquad (7.1)$$

Power consumption in CMOS (complementary metal oxide semiconductor) circuits is a combination of static power and dynamic power. Static power is consumed by the dissipation of leakage currents. Dynamic power is the sum of the short circuit dissipation and the switching power consumed while charging and discharging load capacitances. Dynamic power consumption has quadratic dependency on the supply voltage V_{DD} ,

$$P_{dynamic} \propto C V_{DD}^2 f \qquad \text{---} \qquad (7.2)$$

where, C is the switching capacitance and f is the common switching frequency and $P_{dynamic}$ is the dynamic power dissipation.

Assume that an application is required to perform N operations in t seconds. Then, to keep up with the requirement, it should follow the following equation:

$$\frac{N}{IPC_{avg} * f} \leq t \qquad \text{---} \qquad (7.3)$$

IPC_{avg} refers to the average number of instructions issued per second, across the whole application. From this, the idea frequency is:

$$f_{ideal} = \frac{N}{IPC_{avg} * t} \qquad \text{---} \qquad (7.4)$$

$$P_{dynamic} \propto C V_{DD}^2 \frac{N}{IPC_{avg} * t} \qquad \text{---} \qquad (7.5)$$

From Flynn et al. [Flyn99] the total power is:

$$P_{total} = \frac{1}{2} C V_{DD}^2 \frac{N}{IPC_{avg} * t} + I_{leakage} * V + I_{switching} * V \qquad \text{---} \qquad (7.6)$$

7.3.2 POWER SAVING STRATEGIES

From the above equation (7.5), power reduction is possible by: increasing the IPC, lowering the ideal frequency and by reducing the voltage. Among these, voltage reduction and frequency reduction are being implemented in the modern mobile microprocessors. The operating speed of a circuit depends on the supply voltage as shown above in the equations. Voltage reduction by a factor of S reduces the dynamic power consumption by a factor of S^2. This has been the reason for the popularity of scaling voltage in low power processors.

Power savings strategies can be classified as *Static power management* (SPM) techniques (off-line) such as synthesis and compilation for lower power. *Dynamic power management* (DPM) techniques (on-line) use runtime behavior to reduce power when systems are serving light workloads. Static power management models are used in architecture level such as optimization of the chip design. Many simulators and measurement tools exist to analyze the processor and system power consumption. Among these Wattch [Brtm00], SimplePower [Yvki00], and ARMulator [W3arm] are popular simulators. PowerScope [Flis99] is a power measurement tool to map energy consumption to program structure by augmenting the information gathered by *time-driven statistical sampling.* Dynamic power reduction techniques utilize the runtime behavior of applications to reduce power when the components are serving light workloads or idle. DPM uses Dynamic Voltage Scaling (DVS), and shutting down unused I/O devices. At cluster level, it is used to shut down the unused nodes. Both SPM and DPM techniques can be applied at CPU level and system (non-CPU components such as memory, NIC, disk etc.) level.

Dynamic voltage scaling (DVS) has been proposed and being deployed widely [Bubr00, Intx03, Tran03]. DVS allows a processor to dynamically change speed and voltage at runtime. This utilizes the variance in processor utilization, lowering the voltage (in turn frequency) when processor is lightly loaded and running with maximum frequency when the processor is heavily loaded. This can be extended to all other devices, such as memory, network interface card (NIC) based on their variation of utilization.

In our research, we plan to identify active and inactive devices in executing an application. This can be done using profiling and modeling the energy consumption behavior during the execution time. We plan to schedule the power saving modes for these devices to reduce overall power consumption. The challenge however is to maintain the performance. The whole performance should not be reduced significantly due to power saving schedules. This requires metrics to define the energy savings and performance of the application. In the next section, we discuss the energy-performance metrics in detail.

7.3.3 ENERGY – PERFORMANCE METRICS

The power reduction methods mentioned in the previous section saves power as well as reduces the speed of the components. This directly results in the increase of execution time (loss of performance). This would cause panic in the minds of high performance zealots. But given two algorithms *A* and *B*, with energy and execution time (E_A, T_A) and (E_B, T_B) respectively, the question arises on how to combine both energy and time. The energy-time product E*T was being used as a reasonable metric of energy efficiency

[Goho96]. Martin, Nystroem and Penzes [Manp01] show that E*T product is not the right metric and proposed ET^2 as a special case of the ET^n metric that is voltage independent.

The discussion to find "the" metric that represents energy-performance still continues. Research in [Flyn99, Brmb00] suggests that operating frequency is roughly proportional to the supply voltage. That makes the above equation,

$$P \propto V^3 \propto f^3 \qquad \text{---} \qquad (7.7)$$

This leads to the conclusion that to reduce the power dissipation of a processor designed to operate at high frequency: reduce the voltage (and hence the frequency). There is a limit however to what level V_{DD} can be reduced, depending on the manufacturability and circuit reliability issues. In SIGMETRICS '01 tutorial Bose et al. [Bomd01] suggests to use E or ET^2 type metrics, depending on the class of processors being compared. The caveat is that in future processors, the leakage power control techniques will be used, that use lots of low voltage transistors.

In exposing more about the caveat mentioned above, Lee et al. [Lfdd03] emphasizes to consider the growing portion of leakage and switching current. All the previous research concentrated on reducing the dynamic power specified in equation 7.5, but with the new process technology of sub-0.13-μm, the leakage and switching currents cannot be ignored. This paper also mentions that ET product for the whole system does not reflect the energy savings, when the savings are applied *only* to a particular functional block (L1 cache, NIC etc.). This shows that there is need to find new metrics, which reflect the energy-performance tradeoff.

The goal of any energy-time tradeoff metric is to make energy and time equally important entities. Therefore, we propose that a metric $E^{\alpha} * T^{\beta}$, where α and β can adjust

their values based on the importance of energy-delay. We think that value for α and β depend on the ratio of static and dynamic energies. We have to investigate further on how to find these values.

7.3.4 OPTIMIZING ENERGY – PERFORMANCE

In analyzing the energy savings of a parallel (cluster) machine requires classifying the consumption of all the devices in that machine. In order to combine these two aspects, in this section we derive metrics of total energy consumption of a cluster and the performance of an application. There are many devices in a system where power reduction is possible. Among these, CPU, memory, disk and network card have been of interest since they are equipped with sufficient interface to access them to dynamically from the source code. Assuming that we have a cluster with *N* nodes and each node is equipped with single processor (considering the recent nodes have multiple CPUs). If the power required for the total cluster is $P_{cluster}$, the energy required executing an application from time t_1 to t_2 is,

$$E_{cluster} = \int_{t_1}^{t_2} P_{cluster}\, dt \qquad \text{---} \qquad (7.8)$$

The total power consumed by the cluster is the combination of power consumed by the number of nodes used and the power consumed by the network cards. Power consumed in a node is the total power consumed by all the devices.

$$P_{cluster} = \sum_{i=1}^{N} P(node_i) \qquad \text{---} \qquad (7.9)$$

$$P(node_i) = \sum_{j=1}^{D} P(device_j^i) \qquad \text{---} \qquad (7.10)$$

$P(node_i)$ is the power required by node *i*, and the cluster has *N* nodes. Assuming that there are *D* devices in each node, $P(device_j^i)$ is the power consumed by device *j* in node *i*.

When we apply the optimizations to these individual devices using the voltage scaling or frequency scaling, the energy consumed by each device is a combination of their energy in each state for the amount of time it spent in each state respectively and the amount of energy required for transitions between states. For example if there are two states, the total energy consumed by a device is the sum of energy spent in state 1, energy spent in state 2 and the energy consumed for each transitioning between these states. The transition energy varies based on the states the device moving to. So, the total transition energy is:

$$ET(device_j^i) = \sum_{m=1}^{T} ET(device_{j,m}^i) \quad \text{---} \quad (7.11)$$

Where *T* is the number of transitions for $device_j$ in node *i*. The total energy consumed by $device_j$ in node *i* is:

$$E(device_j^i) = \sum_{k=1}^{S} \int_{t1_k}^{t2_k} P(device_{j,k}^i)dt + ET(device_j^i) \quad \text{---} \quad (7.12)$$

In (7.12), $P(device_{j,k}^i)$ is the power consumed by $device_j$ of node *i* in power state *k*. S is the number of power states for $device_j$. $(t2_k - t1_k)$ is the execution time, $device_j$ spent in state *k*. Total energy consumed in a node after source code modifications is:

$$E'(node_i) = \sum_{j=1}^{D} E(device_j^i) \quad \text{---} \quad (7.13)$$

Total energy consumed by the cluster after modification:

$$E'_{cluster} = \sum_{i=1}^{N} E'(node_i) \quad \text{---} \quad (7.14)$$

If the execution time for original application is *T* and that of modified application is *T'*, then our goal is to find minimum value for (using ET2 metric) $E'_{cluster} * T'^2$ such that

$$(E'_{cluster} * T'^2) \leq (E_{cluster} * T^2) \ \& \ E'_{cluster} \leq E_{cluster} . \qquad \text{---} \qquad (7.15)$$

If we use our proposed metric $E^{\alpha} * T^{\beta}$, the equation 7.15 transforms into the following:

$$(E'^{\alpha}_{cluster} * T'^{\beta}) \leq (E^{\alpha}_{cluster} * T^{\beta}) \ \& \ E'_{cluster} \leq E_{cluster} \qquad \text{---} \qquad (7.16)$$

where, α and β can adjust their values based on the importance of energy-delay and the ratio of static and dynamic energy consumption. We have to investigate further in finding values for α and β.

The next challenge is to balance the performance optimizations and energy saving. As described above the current view of researchers has been the effect of the energy savings on the performance. We need to view the problem in the other direction, how the performance optimizations affect the energy consumption. For example, optimization of improving the locality of data references increases the load on CPU, as there would not be as many cache misses as with unoptimized execution. Another example is overlapping of computation with communication makes both CPU and network card (NIC) busy. This view opens discussion for how the energy consumption changes for each optimization. Do we need to profile and measure the power every time an optimization method is applied? If we measure that for every optimization, we end up with the same problem as we had with finding an effective set of memory performance optimizations. To our best knowledge, this problem is still open and has not been addressed by anyone. [Cfgb03] has proposed a prototype of low-power network supercomputer. A good solution would be to be able to predict the effect of each optimization method on energy consumption.

With our current knowledge of memory performance, we plan attack this problem in predicting this effect, with a highly parameterized model and historical data of previous history of optimizations. The next goal is to find the balance between energy savings and performance using the requirement by equation 7.16.

7.4 SUMMARY

In this chapter, we discuss three areas of extending our fundamental research models. First, data push architecture is applicable to parallel computing clusters and shared memory parallel machines. Many nodes in parallel computers are idle. By using these idle nodes as servers, which provide data push service (DPS-P), data accesses from computing nodes are monitored and predicted data is pushed to the client nodes. These pro-active memory servers have enormous potential to bridge the divergence gap. Second, we have presented the design of File Access Server (FAS) to improve the performance of I/O performance in HEC by using push server architecture. A server can be placed in parallel system to observe the patterns, to predict future data accesses, and to push the data to the client nodes. This idea is innovative and gaining popularity in HEC community. Lastly, we discussed the basics of energy consumption, energy-performance tradeoffs, and the strategies to reduce the performance. We discussed the present dilemma over the metrics and proposed a new metric to find a balance between performance-energy savings tradeoff. All these extensions are based on our data access models. We plan to explore these systems in the future to materialize by testing the real application environment.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

We conclude this dissertation by stating the importance of solving data access latency and how our research effort has impacted in that direction. We present our future research ideas in finding energy-memory performance tradeoff and I/O performance improvement.

8.1 SUMMARY OF CONTRIBUTIONS

Over the last two and half decades, the gap between processor performance and data access performance has been growing wider. This gap has reached to a point, where sustained system performance of current High End Computing machines is only a small fraction of peak performance. This dismal performance situation must be solved in the direction of filling the divergence gap by developing hardware level and software level strategies in computer architecture and in parallel application developing.

Research efforts in solving data access latency problem are numerous. Caching and prefetching are obvious additions made to memory subsystem. Various studies proposed strategies to improve cache utilization by reordering data access patterns of applications, to predict future cache misses, and to prefetch data before a processor needs it. However, many prefetching strategies are limited by the complexity of accurate future data access prediction algorithms within a processor. With the recent revolutions in processor architecture, abundant computing power is available to share the burden of data access by separating computing and data access operations. Some research efforts try to utilize helper threads to pre-execute codes to support this separation. However, existing methods of data access separation are pull-based strategies, where a processor or a helper thread

pulls data for another processor. This leaves burden either on program developers and/or on compilers to perform prefetching decisions, to provide prefetching hints, and to synchronize prefetching with processing. The same applies to software level optimizations in improving cache memory reuse efficiently. Many loop optimization methods have been proposed, but the burden is again left on compilers or developers to perform the optimizations. Application of these optimizations is at middleware level is also a more difficult problem due to the lack of models to optimize data access performance automatically.

The work described in this dissertation concentrates in solving data access performance problem. We designed server-based data push architecture to use complex and adaptive strategies to increase the accuracy of future data access pattern prediction. We have developed analytical data access cost prediction models to automate the process of data access optimization. We have applied this model to improve the performance of message passing interface library. The simulation results of our data push architecture show significant performance gains. This architecture can be applied at various levels of memory hierarchy in order to solve the gap between peak and sustained performance.

In Chapter 3, we explained the architecture of our Data Push Server (DPS). We addressed the issues of monitoring data access history, making spatial and temporal access pattern predictions, and architecture modifications to push the predicted data values close to processing cores. DPS chooses prediction strategies adaptively based on data access history pattern. As there is no single universal algorithm to predict all patterns, our adaptive selection strategy improves the accuracy in prediction of future data accesses. DPS uses prediction algorithms to perform timely pushing of data, in order

to reduce cache pollution. DPS can serve multiple client cores and improves parallelism. We have discussed the modified memory reference operation by CPU and the feasibility of implementing DPS on IBM Cell processor by using SPE local store's software controllability feature.

We presented our Simple Memory Access Cost (SMAC) prediction model in Chapter 4, which predicts data access performance based on data access patterns. The data accesses are classified based on the strides between successive accesses and the size of each contiguous chunk of data. The simplicity of this model is useful to apply various cache performance improving optimizations automatically, avoiding the burden on application developers.

Our work in improving the performance of derived datatypes of MPI implementation is presented in Chapter 5. We discuss a method to quantify data access cost from network communication and middleware latency. This has led to development of memoryLogP model. We apply our SMAC model in improving the performance MPI derived datatypes by selecting optimization parameters for loop optimizations, such as cache blocking, loop unrolling, software prefetching, array padding. These parameters are passed to optimized templates to pack/unpack data before sending/receiving between processes. The performance improvement is substantial and has direct impact on parallel application development.

Chapter 6 presents the simulation results of DPS architecture. We discuss the modifications to SimpleScalar simulator in order to implement DPS architecture. The results show a great effect on data access performance for various SPEC CPU2000 benchmarks, which have high cache miss rates.

Applying DPS architecture at various levels of memory hierarchy has potential to improve data access performance further, especially at I/O level. In Chapter 7, we discussed the designs and initial results of using DPS model at memory server level and I/O level. DPS for parallel computing (DPS-P) serves client nodes in a cluster environment or SMP machine level by observing the data access patterns of the clients and pushing data closer to processing in time. We also presented the basic metrics of another important problem, finding balance between energy consumption and memory performance improvement.

8.2 IMPACT

In parallel application development, aside from poor data access performance that contributes to the gap between peak and sustained performances, there is another non-technical gap which is often ignored by researchers. Even though many of advanced architectures offer various optimizations, application developers from various domains of sciences are not fully aware of these optimizations. This expertise gap in parallel programming development must be addressed in order to improve the productivity of current HEC. Our work has demonstrated the need for new strategies in computer architecture and practical solutions for optimizing the data access performance automatically. Evolving architectures of general-purpose processors provide great opportunity for compute intensive workloads. Many scientific workloads, however, are typically data intensive than compute intensive. The performance of data intensive workloads is limited by data access latency. Our work is directed towards these applications. We developed prefetching and automatic cache optimization methods in our

research in this dissertation. These strategies have a broad impact in developing systems that avoid burden on application developers and provide superior data access performance, automatically.

Our work in improving the performance of MPI derived datatypes will be in the future release of MPICH2 implementation. Previous research efforts were implemented on MPI derived datatypes to bring their performance to a level a naïve developer can achieve. Our implementation improves this performance further, which an advanced developer can obtain by applying various cache optimizations to pack and unpack data in data communication. This strategy directly attempts our goal of providing superior performance for novice and other non-computer science parallel application developers. This work can be further improved to apply to general applications.

8.3 FUTURE WORK: ENERGY-PERFORMANCE TRADEOFF

Technical advances in processor and chip technologies also pose increased power consumption is a problem to modern computing. Following the Moore's law, the current improvement of chip performance depends on the raising of number of transistors. This increases the power demands rapidly caused by numerous devices in the current supercomputers as the high end computing machines use thousands of processors. The increasing power requirement can be understood by comparing the consumption of Intel Pentium 4 (75 watts) to Intel Itanium (130 watts). The current fastest supercomputer BlueGene/L will use 1.5 megawatts of power per day when it is fully operational. This is a significant figure given that one megawatt is enough to power about 800 homes per day [Yeom04]. Multi-core processor architectures are attempting to achieve low power

designs. However, with a requirement for large supercomputers, the power consumption problem is growing rapidly. With the scale of the latest supercomputers is increasing, it is necessary to save power wherever possible with minimum performance compromise.

There are a few existing power saving schemes. Operating system level schemes turn processor, disk and monitor off when the user doesn't use a computer for a specific amount of time. Much Research effort has been spent on low power alternatives for battery-powered devices at hardware and software level [Bubr00, Intx03, Tran03, Brtm00, Sibd99]. Dynamic voltage scaling (DVS) is a technique for exploiting hardware capabilities to select an appropriate clock rate and voltage to meet application requirements at the lowest energy cost. Many chips are coming out with this feature and many DVS algorithms have been proposed [Wwds94, Gocw95, Pebb98, Glfm00, Flrm01, Pols01]. Reducing CPU voltage reduces the energy consumption substantially. Extending this trend towards high end computing, processor manufacturers recognized the necessity to provide software level interfaces to control the power saving levels. Energy is proportional to the square of frequency; so reducing the frequency by half reduces the energy consumption to a quarter of what is needed at full frequency. But it is the responsibility of the application developers to design high performance applications by keeping these energy saving schemes in mind.

Our memory performance optimization can be extended towards developing models for energy-performance tradeoff and automatic optimization of performance and reduction of energy consumption. In Chapter 7, we have presented the basics towards developing new energy-performance models. We plan to expand our research in this direction to solve this important problem.

8.4 FUTURE WORK: HIGH-END COMPUTING I/O

While memory access performance has been getting a lot of attention, I/O access performance has been continuously ignored by many researchers. In our future research, we aim to reduce the performance disparity between processors and I/O access to and from disks, ultimately to reduce the divergence gap. The main reason behind poor I/O performance has again been the gap between the improvements of processor performance and storage performance. Our goal is to reduce this gap by overlapping data movement from disks and computing efficiently.

In solving the I/O problem, parallel file systems such as Lustre [Cfsi00], GPFS [Scha02], PanFS [Nase04], PVFS [Clrr00], PPFS2 [Trre04] provide high bandwidth for simple I/O access patterns. However, for complex non-contiguous access patterns the performance of these file systems is relatively poor, which directly wastes processor cycles. Numerous studies of the I/O characteristics of parallel applications (such as weather forecasting, seismic exploration, climate modeling, and bioinformatics applications) have shown that these applications make large number of requests for small and noncontiguous pieces of data from files [Nkpe96, Cacr95, Smre98]. High-level I/O libraries such as HDF-5 [HDF5] also end up making lots of small requests. Although techniques such as data sieving and collective I/O [Thgl02] can be used in some cases to merge small I/O requests into large ones, it is not possible to eliminate small I/O requests entirely. Therefore, improving the performance of small I/O requests is a necessity to achieve the vision of having systems with balanced compute and I/O capability.

Among various strategies to improve I/O performance, caching and prefetching are considered to be effective. Large buffers in memory (referred to as I/O caches) are provided to store data blocks that are frequently used, instead of accessing disk for each request. With prefetching, data is fetched to these I/O caches before an application requests that data. In this way, file access latency is overlapped with computation. However, existing usage of I/O caches, prefetching strategies, and file systems to optimize I/O performance is conservative and limited to static prediction strategies due to tradeoff between the amount of access pattern information retained and the achievable resolution in prefetching decisions. To solve this problem and to overlap the file access latency effectively, novel adaptive strategies are necessary.

With recent advances in processor and network technologies, it is time to explore more adaptive and aggressive strategies in moving data closer to the processing unit. Emerging HEC machines with hundreds of thousands of processors with multiple cores provide massive computing power. Latest network technologies such as Infiniband are aiming to provide nanosecond latencies. Utilizing these advances in improving the I/O performance will reduce time-to-solution and in turn reduce the gap between peak performance and sustained performance.

With the experience in designing DPS, we plan to develop an intelligent File Access Server (FAS) to reduce I/O access time. This server pro-actively “pushes” data *in time* in client’s memory. We have presented the design diagrams of FAS and initial results in Chapter 7 of this dissertation. As our next step, we plan to implement FAS on PVFS.

8.5 SUMMARY

This dissertation has presented the need to develop novel architectures for HEC application workloads that are typically data intensive. We have provided designs of server-based data push architecture for memory subsystem and developed new models to improve the cache performance dynamically. These models were applied to improving the performance of MPI library and can be applied to optimize cache performance of many scientific applications. The data push server approach has demonstrated improved overall application performance for benchmarks with high L1 cache misses. Our server-based push strategy has enormous potential to be applied at various levels of memory hierarchy including shared memory and disk I/O.

During this dissertation, I have conducted research to improve the data-access performance. This research can be continued further to attack various unsolved problems in reducing energy consumption, optimizing data access performance automatically for various parallel libraries, improving I/O performance of parallel file systems etc. I plan to carry on my research efforts along these directions and look forward to solving more problems.

BIBLIOGRAPHY

[Abra03] Spencer Abraham, “Facilities for the Future of Science: A Twenty-Year Outlook”, DOE Office of Science report, 2003.

[Acks99] Anurag A, Samir K, and Sanjeev S, “Dodo: A User-level System for Exploiting Idle Memory in Workstation Clusters”, In Proceedings of the Eighth IEEE Intl. Symposium on High Performance Distributed Computing 8, Aug. 1999.

[Aiss95] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model---One Step Closer Towards a Realistic Model for Parallel Computation", In proceedings of Symposium of Parallel Algorithms and Architectures (SPAA), pp. 95-105, July 1995.

[Alke96] T. Alexander and G. Kedem, “Distributed predictive cache design for high performance memory system”, In Proceedings of the 2nd International Symposium on High Performance Computer Architecture (HPCA), pages 254--263, 1996.

[Alpb02] Ahmed Amer, Darell Long, Jehan-Francios Paris, and Randal Burns, “File access prediction with adjustable accuracy”, International Performance Conference on Computers and Communication, Phoenix, AZ, April 2002.

[Anpd01] M. Annavaram, J. M. Patel, and E. S. Davidson, “Data prefetching by dependence graph pre-computation”, In Proceedings of the 28th ISCA, pages 52-61, 2001.

[Baer88] Baer, J.-L. and Wang, W.-H., “On the inclusion properties for multi-level cache hierarchies”, In Proceedings of the 15th Annual International Symposium on Computer Architecture, 73-80, 1988.

[Bail02] D. H. Bailey, “21st Century High-End Computing,” In invited Talk of Application, Algorithms and Architectures workshop for BlueGene/L, 2002.

[Bawu96] S. Baylor and C. Wu, “Parallel I/O Workload Characteristics Using Vesta”, In R. Jain, J.Werth, and J. Browne, editors, Input/Output in Parallel and Distributed Computer Systems, chapter 7, pages 167–185. Kluwer Academic Publishers, 1996.

[Bckk95] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny, “A model and compilation strategy for out-of-core data parallel programs”, In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), pages 1--10. ACM Press, July 1995. ACM SIGPLAN Notices, Vol. 30, No. 8.

[Bgst03] Surendra Byna, William Gropp, Xian-He Sun, and Rajeev Thakur, “Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost,” IEEE International Conference on Cluster Computing, 2003, Hong Kong, December 2003.

[Bgst031] Surendra Byna, William Gropp, Xian-He Sun, and Rajeev Thakur, “Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost”, Poster at IEEE Supercomputing 2003 (SC ’03) (Won Best Poster Award).

[Bocr96] Rajesh Bordawekar, Alok Choudhary, and J. Ramanujam, “Automatic optimization of communication in compiling out-of-core stencil codes”, In Proceedings of the 10th ACM International Conference on Supercomputing, pages 366--373, Philadelphia, PA, May 1996, ACM Press.

[Bogr94] G. E. P. Box, G. M. Jenkins, and G. C. Reinsel, “Time Series Analysis: Forecasting and Control”, 3rd ed. Prentice Hall, 1994.

[Bomd01] Pradip Bose, M. Martonosi, and David Brooks, “Modeling and Analyzing CPU Power and Performance:Metrics, Methods, and Abstractions,” SIGMETRICS 2001 / Performance 2001 - Tutorials, 2001.

[Brmb00] D. Brooks, M. Martonosi and P. Bose, “Power-aware microarchitecture: design and modeling challenges of the next generation microprocessors,” Proc. Power Aware Computer Systems (PACS) Workshop (in conjunction with ASPLOS), Nov. 2000; reprinted in Lecture Notes on Computer Science (LNCS).

[Brms95] Peter Brezany , Thomas A. Mück , Erich Schikuta, “Language, compiler and parallel database support for I/O intensive applications”, Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, p.14-20, May 03-05, 1995.

[Brtm00] D.Brooks, V.Tiwari, and M.Martonosi, “Wattch: A Framework for Architectural-Level Power Analysis and Optimizations”, In Proceedings of the 27th International Symposium on Computer Architecture (ISCA), June 2000.

[Bsgt04] Surendra Byna, Xian-He Sun, William Gropp and Rajeev Thakur, “Predicting Memory-Access Cost Based on Data-Access Patterns,” In proceedings of IEEE International Conference on Cluster Computing, 2004, San Diego, September 2004.

[Bstg06] Surendra Byna, Xian-He Sun, Rajeev Thakur and William Gropp, “Automatic Memory Optimizations for Improving MPI Derived Datatype Performance”, selected for publication in Proceedings of the 13th European PVM/MPI Users' Group Meeting, 2006 (Euro PVM/MPI '06).

[Buab96] D.C. Burger, T.M. Austin, and S. Bennett, “Evaluating Future Microprocessors: the SimpleScalar Tool Set”, University of Wisconsin-Madison Computer Sciences Technical Report 1308, July 1996.

[Bubr00] T. Burd and R. Brodersen, “Design issues for dynamic voltage scaling”, In Proc. International Symposium on Low Power Electronics and Design, pages 9–14, July 2000.

[Bycs02] Surendra Byna, Kirk W. Cameron and Xian-He Sun, “Memory-Aware Communication – An Experimental Study with MPI,” The 1st International Workshop on Hardware/Software Support for Parallel and Distributed Scientific and Engineering Computing (SPDSEC02), September, 2002. (Charlottesville, VA)

[Bycs03] Surendra Byna, Kirk Cameron, and Xian-He Sun, “Quantification of memory communication,” Book Chapter in High Performance Scientific and Engineering Computing- Hardware/Software Support, Kluwer Academic Publishers, Chapter 3, pp: 31-44, 2003.

[Bycs04] Surendra Byna, Kirk W. Cameron and Xian-He Sun, “Isolating Costs in Shared Memory Communication Buffering”, Parallel Processing Letters, Volume 14, Issue 2, June 2004.

[Bysc06] Surendra Byna, Xian-He Sun, and Yong Chen, “Server-based Data Push for Multi-processor Environments”, IIT CS TR-2006-031, September 2006

[Bysu06] Surendra Byna, Xian-He Sun, and Ryan Nakhoul, “Memory Servers: A Scope of SOA for High-End Computing”, In proceedings of IEEE International Conference on Services Computing, 2006 (SCC '06).

[Cacr95] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed, “Input/Output Characteristics of Scalable Parallel Applications,” Proceedings of SC ’95, December 1995.

[Cage04] Kirk W. Cameron and Rong Ge, “Predicting and Evaluating Distributed Communication Performance”, In proceedings of the 16th High Performance Computing, Networking and Storage Conferece (SC 2004), November, 2004.

[Casu03] Kirk W. Cameron, and Xian-He Sun, “Quantifying Locality Effect in Data Access Delay: Memory logP,” in Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03), April 2003.

[Cbeh06] IBM Cell Broadband Engine Architecture group, “SPU C/C++ Language Extensions”, CBEA Joint Software Reference Environment (JSRE) Series, March, 2006.

[Cffh95] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong, “Overview of

the MPI-IO Parallel I/O Interface", In Proc. of the Third Workshop on I/O in Parallel and Distributed Systems, Santa Barbara, CA, April 1995. IPDPS '95.

[Cfgb03] Kirk W. Cameron, Xizhou Feng, Rong Ge, and Duncan Buell, "The Argus Prototype: Aggregate use of load modules as a low-power network supercomputer", USC CSCE TR-2003-019. September, 2003.

[Cfsi00] Cluster File Systems Inc., "Lustre: A scalable, high performance file system", Whitepaper, http://www.lustre.org/docs/whitepaper.pdf

[Chat00] S. Chatterjee and S. Sen, "Cache-Efficient Matrix Transposition", Proceedings of the 6th International Symposium on High-Performance Computer Architecture, Toulouse, France, January 2000, pages 195-205.

[Chat01] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck, "Exact Analysis of the Cache Behavior of Nested Loops", Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, Snowbird, UT, June 2001.

[Chba95] T.F. Chen and J.L. Baer, "Effective Hardware-Based Data Prefetching for High Performance Processors," IEEE Transactions on Computers, pp. 609-623, 1995.

[Chuh04] I-Hsin Chung, Jeffrey K. Hollingsworth, "Using Information from Prior Runs to Improve Automated Tuning Systems", Proceedings of SC'04, Nov. 2004

[Ckps96] D. E. Culler, R. Karp, D. A. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken, "LogP: A Practical Model of Parallel Computation," Communications of the ACM, vol. 39, pp. 78-85, 1996.

[Clea06] ClearSpeed Technology, “CSX Processor Architecture Whitepaper”, http://www.clearspeed.com/optimize.html

[Clrr00] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, “PVFS: A Parallel File System For Linux Clusters'”, Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, October 2000, pp. 317-327.

[Cogr90] D. Comer and J. Griffioen, “A New Design for Distributed Systems: The Remote Memory Model”, In The Proceedings of the 1990 Summer USENIX Conference, pages 127-136. USENIX Association, June 1990.

[Cogu96] J. H. Conway and R. K. Guy, “The Book of Numbers”, Springer-Verlag, New York, 1996, ISBN: 038797993X.

[Csbr05] Kirk W. Cameron, Xian-He Sun, Surendra Byna and Rong Ge, “Predicting and Evaluating Memory Communication Performance,” submitted to IEEE Transactions on Parallel and Distributed Systems, February 2005.

[Ctws01] J. Collins, D. Tullsen, H. Wang, and J. Shen, “Dynamic speculative precomputation”, In Proceedings of the International Symposium on Microarchitecture, 2001.

[Cull97] D. Culler, J.P. Singh, and A. Gupta, “Modern Parallel Computer Architecture”, Book, Morgan Kaufmann Publishers, 1997.

[Cusg98] D. Culler, J.P. Singh, and A. Gupta, “Parallel Computer Architecture: A Hardware/Software Approach”, Morgan Kaufmann, August 1998, ISBN 1558603433.

[Dads93] F. Dahlgren, M. Dubois, and P. Stenström, “Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors,” Proc. 1993 Int’l Conf. Parallel Processing, CRC Press, Boca Raton, Fla., 1993, pp. I56-I63

[Dads95] F. Dahlgren, M. Dubois, and P. Stenström, “Sequential Hardware Prefetching in Shared-Memory Multiprocessors”, IEEE Transactions on Parallel and Distributed Systems, Volume 6, Issue 7, July 1995, pp. 733 - 746

[Dcmd90] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff, “A set of Level 3 Basic Linear Algebra Subprograms”, ACM Transactions on Mathematical Software, 16(1):1--17, 1990.

[Derr99] Luiz DeRose and Daniel A. Reed, “SvPablo: A multi-language architecture-independent performance analysis system”, In Proceedings of the International Conference on Parallel Processing, Fukushima, Japan, September 1999.

[Dhpcs] DARPA, High Productivity Computing Systems (HPCS), Vision: Focus on the Lost Dimension of HPC-“User & System Efficiency and Productivity” http://www.darpa.mil/ipto/programs/hpcs/vision.htm

[Dkkl99] Carole Dulong, Rakesh Krishnaiyer, and Dattatraya Kulkarni “An Overview of the Intel IA-64 Compiler”, 1999, Intel Technology Journal.

[Dowe06] Jack Doweck, “Inside Intel Core Microarchitecture and Smart Memory Access”, White paper, Intel Research website.

[Feza91] E. W. Felten and J. Zahorjan, “Issues in the Implementation of a Remote Memory Paging System”, Technical Report 91-03-09, Department of Computer Science and Engineering, University of Washington, March 1991.

[Flis99] J. Flinn and M. Satyanarayanan, “Powerscope: A tool for profiling the energy usage of mobile applications”, In Proceedings of the Second IEEE, Workshop on Mobile Computing Systems and Applications, 1999.

[Flrm01] K. Flautner, S. Reinhardt, and T.Mudge, “Automatic performance setting for dynamic voltage scaling”, In 7th Annual International Conference on Mobile Computing and Networking, pages 260–271, 2001.

[Flyn99] M.J. Flynn et al., “Deep-Submicron Microprocessor Design Issues,” IEEE Micro, Vol. 19, No. 4, July/Aug. 1999, pp. 11-22.

[Fmpk95] M. Freeley, W. Morgan, F. Pighin, A.Karlin, and H. Levy, “Implementing Global Memory Management in a Workstation Cluster”, In Proceedings of the 15th ACM Symposium on Operating Systems Principles, December 1995.

[Fupa91] J. Fu and J.H. Patel, “Data prefetching in multiprocessor vector cache memories”, In Proceedings of the 17th annual International Symposium on Computer Architecture, pp. 54-63, 1991.

[Gabu05] Ilya Ganusov and Martin Burtscher, “Future Execution: A Hardware Prefetching Technique for Chip Multiprocessors”, Proceedings of the 14th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT'05), 2005.

[Ghos99] Somnath Ghosh , Margaret Martonosi , and Sharad Malik, “Cache miss equations: a compiler framework for analyzing and tuning memory behavior”, ACM Transactions on Programming Languages and Systems (TOPLAS), v.21 n.4, p.703-746, July 1999

[Glfm00] D. Grunwald, P. Levis, K. Farkas, C. Morrey, and M. Neufeld, “Policies for dynamic clock scheduling”, In Symposium on Operating Systems Design and Implementation (OSDI), October 2000.

[Gocw95] K. Govil, E. Chan, and H. Wasserman, “Comparing algorithms for dynamic speed-setting of a low-power CPU”, In 1st Annual International Conference on Mobile Computing and Networking, November 1995.

[Goho96] R. Gonzales and M. Horowitz, “Energy Dissipation in General Purpose Microprocessors”, IEEE Journal of Solid-State Circuits, Vol. 31, No. 9, 1996.

[Grap94] J. Griffioen and R. Appleton, “Reducing File System Latency using a Predictive Approach”, In Proceedings of the Summer 1994 USENIX Technical Conference, pages 197–207. USENIX, 1994.

[Grap95] J. Griffioen and R. Appleton, “Performance Measurements of Automatic Prefetching”, In Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems, pages 237–242. IASTED, Sept. 1995.

[Grla99] William Gropp, Ewing Lusk, and Anthony Skjellum, “Using MPI: Portable Parallel Programming with the Message-Passing Interface”, MIT Press, 2nd edition, 1999.

[Grld99] William Gropp, Ewing Lusk, and Deborah Swider, “Improving the Performance of MPI Derived Datatypes”, in Proceedings of the Third MPI Developer's and User's Conference, MPI Software Technology Press, pp. 25–30, March 1999.

[Hafe04] Wessam Hassanein, José Fortes and Rudolf Eigenmann. “Data Forwarding through In-Memory Precomputation Threads”, In Proceedings of the International Conference on Supercomputing (ICS), 2004.

[HDF5] The HDF5 Project, HDF5 - A New Generation of HDF, NCSA, University of Illinois at Urbana Champaign. Available at http://hdf.ncsa.uiuc.edu/HDF5

[Hepa06] John Hennessy and David Patterson, “Computer Architecture: A Quantitative Approach”, Morgan Kaufmann Publishers, 4th Edition, 2006.

[Hire03] T. Highley and P. Reynolds, “Marginal Cost-Benefit Analysis for Predictive File Prefetching”, Proceedings of the 41st Annual ACM Southeast Conference (ACMSE 2003), Savannah, GA.

[Hugh00] C. J. Hughes, “Prefetching Linked Data Structures in Systems with Merged DRAM-Logic”, Master’s thesis, University of Illinois at Urbana-Champaign, May 2000, Technical Report UIUCDCS-R-2001-2221.

[Ibmb04] “IBM Blue Gene Project”, http://www.research.ibm.com/bluegene/

[Ibmc06] IBM, Cell Broadband Engine resource center, http://www-128.ibm.com/developerworks/power/cell/

[Iflp93] L. Iftode, K. Li, and K. Petersen, “Memory servers for multicomputers”, In Proceedings of COMPCON 93, 1993, pp. 538—547.

[Intx03] Intel XScale microarchitecture. http://developer.intel.com/design/intelxscale/.

[Jaco96] B.L. Jacob, “An analytical model for designing memory hierarchies”, IEEE Transaction on Computers, volume 45, pp. 83-105, 1996.

[Jogr97] D.Joseph and D. Grunwald. “Prefetching Using Markov Predictors”, Proceedings of the 24th Annual Symposium on Computer Architecture, Denver-Colorado, pp 252-263, June 2-4 1997.

[Kand99] M. Kandemir, J. Ramanujam and A. Choudhary, “Cache Locality by a Combination of Loop and Data Transformations,” IEEE Transactions on Computers (TC) 48(2): 159–167, February 1999.

[Kasi02] Gokul Kandiraju, and Anand Sivasubramaniam, “Going the Distance for TLB Prefetching: An Application-Driven Study”, In Proceedings of the ISCA 2002.

[Kcky01] N. Kohout, S. Choi, D. Kim, and D. Yeung, “Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes,” In Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques, 2001.

[Kuen94] G. H. Kuenning, “The Design of the SEER Predictive Caching System”, In Proceedings IEEE Workshop on Mobile Computing Systems and Applications, 1994, pages 37–43, IEEE, 1994.

[Kusn05] Dimitri F. Kusnezov, “National Nuclear Security Administration Advanced Simulation and Computing: Ushering in a New Decade of Predictive Capability”, Invited speech at Supercomputing 2005, Seattle.

[Lamm06] LAM/MPI Parallel Computing, http://www.lam-mpi.org/

[Larw91] Monica Lam, Edward E. Rothberg and Michael E. Wolf, “ The cache performance of blocked algorithms”, Proceedings of the Fourth International

Conference on Architectural Support for Programming Languages and Operating Systems, April 1991

[Ledu97] H. Lei and D. Duchamp, “An Analytical Approach to File Prefetching”, In Proceedings of the 1997 USENIX Annual Technical Conference, pages 275–288. USENIX, Jan. 1997.

[Lfdd03] H.-H. S. Lee, J. B. Fryman, A. U. Diril, and Y. S. Dhillon, “The Elusive Metric for Low-Power Architecture Research”, In Workshop on Complexity-Effective Design in conjunction with ISCA-30, 2003.

[Lukc01] Chi-Keung Luk, “Tolerating memory latency through Software-Controlled Pre-Execution in simultaneous multithreading processors”, In Proceedings of the 28th Annual International Symposium on Computer Architecture, pages 40-51, 2001.

[Lwps04] Q. Lu, J. Wu, D. Panda and P. Sadayappan, “Applying MPI Derived Datatypes to the NAS Benchmarks: A Case Study,” Technical Report OSU-CISRC-4/04-TR19, Ohio State University.

[Lwwh02] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen, “Post-Pass Binary Adaptation Tool for Software-Based Speculative Precomputation”, In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’02), 2002.

[Mamt95] B. M. Maggs, L. R. Matheson, and R. E. Tarjan, “Models of Parallel Computation: A Survey ad Synthesis,” in proceedings of 28th Hawaii International Conference on System Sciences (HICSS), Honolulu, HI, 1995.

[Manp01] Alain J. Martin, Mika Nyström, and Paul Penzes, “ET2: A Metric For Time and Energy Efficiency of Computation”, Power-Aware Computing, R.Melhem and R.Graybill ed., Kluwer Academic Publishers, 2001

[Mark87] Hill, M.D. “Aspects of cache memory and instruction buffer performance”, Ph.D. Thesis, University of California, Berkeley, 1987.

[Mcca95] John D. McCalpin, “Memory bandwidth and machine balance in current high performance computers”, IEEE Technical Committee on Computer Architecture, 1995. http://www.cs.virginia.edu/stream.

[Mcch95] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, “The Paradyn parallel performance measurement tool”, IEEE Computer, 28(11):37-46, November 1995.

[Mcki96] K.S. McKinley, S.Carr, and C.W. Tseng, “Improving data locality with loop transformations”, ACM TOPLAS, 18(4): 424-453. July 1996

[Memo04] Nimrod Megiddo, and Dharmendra S. Modha, “ARC: A Self-Tuning, Low Overhead Replacement Cache”, USENIX File and Storage Technologies (FAST), March 31, 2003, San Francisco, CA..

[Mitc01] N. Mitchell, L. Carter, and J. Ferrante, “A modal model of memory”, In proceedings of lectures in Computer Science Springer, editors: V.N.Alexandrov, J.J. Dongarra, May 28-30, 2001.

[Modk96] Todd C. Mowry , Angela K. Demke , Orran Krieger, “Automatic compiler-inserted I/O prefetching for out-of-core applications”, Proceedings of the second USENIX symposium on Operating systems design and implementation, p.3-17, October 29-November 01, 1996, Seattle.

[Mofr98] C. A. Moritz and M. I. Frank, “LoGPC: Modeling Network Contention in Message-Passing Programs,” in proceedings of SIGMETRICS '98, Madison, WI, 1998.

[Mogu91] T. Mowry and A. Gupta, “Tolerating Latency Through Software-controlled Prefetching in Shared-memory Multiprocessors,” Journal of Parallel and Distributed Computing, Volume 12, Issue 2, June 1991.

[Molg92] T.C. Mowry, S. Lam, and A. Gupta, “Design and Evaluation of a Compiler Algorithm for Prefetching,” Proc. Fifth Int’l Conf. Architectural Support for Programming Languages and Operating Systems, ACM Press, New York, 1992, pp. 62-73.

[Moos01] Shirley Moore, and Nils Smeds, “Performance tuning using hardware counter data”, Presentation at SuperComputing 2001, Nov. 01.

[Mpif95] Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard, Version 1.1,” http://www.mpi-forum.org/docs/docs.html, June 1995.

[Mpif96] Message Passing Interface Forum, “MPI-2: Extensions to the Message-Passing Interface”, Technical Report, University of Tennessee, Knoxville, 1996.

[Mpif98] Message Passing Interface Forum, MPI2: A message passing interface standard, High Performance Computing Applications, 12(1-2):1-299, 1998.

[Nase04] A. M. David Nagle, Denis Serenyi, “The Panasas ActiveScale Storage Cluster - Delivering Scalable High Bandwidth Storage”, In Proceedings of Supercomputing ‘04, November 2004.

[Ncsa00] National Center for Supercomputing Applications Archives, (NCSA) “Understanding Performance on the SGI Origin 2000” NCSA Online document, http://archive.ncsa.uiuc.edu/SCD/Perf/Tuning/Tips/Tuning.html.

[Nkpe96] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best, “File-Access Characteristics of Parallel Scientific Workloads”, IEEE Transactions on Parallel and Distributed Systems, 7(10):1075–1089, October 1996.

[Ogma96] H. Ogawa and S. Matsuoka, “OMPI: Optimizing MPI Programs using Partial Evaluation,” In Proceedings of IEEE/ACM Supercomputing Conference, Pittsburgh, November 1996.

[Olxl04] John Oleszkiewicz ,Li Xiao, and Yunhao Liu, "Parallel Network RAM: Effectively Utilizing Global Cluster Memory for Large Data-Intensive Parallel Programs", 33th International Conference on Parallel Processing (ICPP 2004), Montreal, Quebec, Canada, August, 2004.

[Patt96] D. A. Patterson and J. L. Hennessy, “Computer Architecture: A quantitative approach”, 2nd edition. San Fancisco, CA: Morgan Kaufmann Publishers, 1996.

[Peak98] Y. Paek, J. Hoeflinger, and D. Padua, “Simplification of Array Access Patterns for Compiler Optimizations”. In Proceedings of the ACM SIGPLAN 98 Conference on Programming Language Design and Implementation, June 1998.

[Pebb98] Trevor Pering, Thomas D. Burd, and RobertW. Brodersen, “The simulation and evaluation of dynamic scaling algorithms”, In International Symp. on Low Power Electronics and Design (ISLPED), August 1998.

[Pggs95] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, “Informed Prefetching and Caching”, In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP ’95), ACM, 1995.

[Pols01] J. Pouwelse, K. Langendoen, and H. Sips, “Dynamic voltage scaling on a low-power microprocessor”, In The Seventh Annual International Conference on Mobile Computing and Networking 2001, pages 251–259, 2001.

[Prra99] V. S. Pai, P. Ranganathan, H. Abdel-Shafi and S. Adve. “The Impact of Exploiting Instruction-Level Parallelism on Shared-Memory Multiprocessors”, IEEE Transactions on Computers, 48(2):218--226, February 1999.

[Pirc95] Charles Price, “MIPS IV Instruction Set, revision 3.1”, MIPS Technologies, Inc., Mountain View, CA, 1995.

[Reth00] Ralf Reussner, Jesper Larsson Träff, and Gunnar Hunzelmann, “A Benchmark for MPI Derived Datatypes,” In Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users’ Group Meeting, volume 1908 of Lecture Notes in Computer Science, pages 10-17, 2000.

[Rivt99] Gabriel Rivera and Chau-Wen Tseng, “Locality optimizations for multi-level caches,” in Proceedings of Supercomputing 1999: High-Performance Networking and Computing, November 1999.

[Romg03] R. Ross, N. Miller, and W. Gropp, “Implementing Fast and Reusable Datatype Processing,” In Recent Advances in Parallel Virtual Machine and Message Passing Interface, 10th European PVM/MPI Users’ Group Meeting, volume 2840 of Lecture Notes in Computer Science, pages 404-413, 2003.

[Roms98] A. Roth, A. Moshovos, and G. S. Sohi, “Dependence based prefetching for linked data structures”, In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 115–126, 1998.

[Roso01] Amir Roth and Gurindar S. Sohi, “Speculative data-driven multithreading”, In Proceedings of the 7th International Symposium on High Performance Computer Architecture, 2001.

[Ryan02] Rong Yan and Seth C, “Goldstein Mobile Memory: Improving Memory Locality in Very Large Reconfigurable Fabrics”, FCCM '02, Napa Valley, CA, April 2002.

[Saav95] Rafael H. Saavedra and Alan Jay Smith, “Measuring Cache and {TLB} Performance and Their Effect on Benchmark Runtimes”, IEEE Transactions on Computers, Volume: 44, number: 10, p1223-1235, 1995.

[Safs00] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström, “Recency-based TLB Preloading,” in Proceedings of the 27th Annual International Symposium on Computer Architecture, pp. 117–127, June 2000.

[Sagc93] R. H. Saavedra , R. S. Gaines , M. J. Carlton, “Microbenchmark analysis of the KSR1”, in Proceedings of the 1993 ACM/IEEE conference on Supercomputing, p.202-213, December 1993, Portland, Oregon.

[Scha02] F. Schmuck and R. Haskin, “GPFS: A Shared-Disk File System for Large Computing Clusters”, In First USENIX Conference on File and Storage Technologies, pages 231--244. USENIX, Jan. 2002.

[Sech00] S. Sen and S. Chatterjee, “Towards a theory of Cache efficient algorithms”, SODA, 2000

[Shpc01] T. Sherwood, E. Perelman, and B. Calder, “Basic block distribution analysis to find periodic behavior and simulation points in applications”, In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, pages 3–14, 2001.

[Sibd99] T.Simunic, L.Benini, and G. De Michelli, “Energy-Efficient Design of Battery- Powered Embedded Systems”, Int’l Symposium on Low Power Electronics and Design, 1999.

[Skte05] B. Sinharoy, R.N. Kalla, J.M Tendler, R.J. Eickemeyer, and J.B. Joyner, “Power5 System Microarchitecture”, IBM Journal of Research And Development, Vol. 49, No. 4/5, July-September 2005.

[Smit82] James E. Smith, “Decoupled access/execute computer architectures”, In Proceedings of the 9th annual International Symposium on Computer Architecture (ISCA), p.112-119, 1982.

[Smre98] Evgenia Smirni and Daniel A. Reed, “Lessons from Characterizing the Input/Output Behavior of Parallel Scientific Applications,” Performance Evaluation, 1998 , Volume 33, pp. 27-44.

[Sohi90] G. S. Sohi, “Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers”, IEEE Transaction of Computer, 39(3):349--359, 1990.

[Solt02] Y.Solihin, J.Lee, and J.Torrellas “Using a User-Level Memory Thread for Correlation Prefetching”, In Proceedings of International Symposium on Computer Architecture, May 2002, 171-182.

[Spec00] Standard Performance Evaluation Corporation, SPEC Benchmarks, http://www.spec.org/.

[Star00] Eugene W. Stark, “SAMSON: Network Memory Server Project”, http://bsd7.starkhome.cs.sunysb.edu/~samson/

[Suby05] Xian-He Sun and Surendra Byna, “Data-access Memory Servers for Multi-processor Environments”, IIT CS TR-2005-001, November 2005, http://www.cs.iit.edu/~suren/research.html

[Tefj98] O. Temam, C. Fricker, and W. Jalby, “Cache Awareness in Blocking Techniques,” Journal of Programming Languages, 1998.

[Thgl02] Rajeev Thakur, William Gropp, and Ewing Lusk, “Optimizing Noncontiguous Accesses in MPI-IO,” Parallel Computing, (28)1:83-105, January 2002.

[Thgl99] Rajeev Thakur, William Gropp, and Ewing Lusk, “On Implementing MPI-IO Portably and with High Performance”, in Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems, May 1999, pp. 23-32.

[Thrz99] Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann, “Flattening on the Fly: Efficient Handling of MPI Derived Datatypes,” in

Proceedings of the 6th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science, Vol. 1697, Springer, pp. 109–116, 1999.

[Tran03] Transmeta Crusoe microarchitecture. http://www.transmeta.com.

[Trre04] Nancy Tran, Daniel A. Reed. “Automatic ARIMA Time Series Modeling for Adaptive I/O Prefetching,” IEEE Transactions on Parallel and Distributed Systems, vol. 15, no. 4, pp. 362-377, April, 2004.

[Vali00] Steven P. VanderWiel and David J. Lilja, “Data prefetch mechanisms”, ACM Computing Surveys, 32(2):174--199, June 2000.

[Vali90] L. G. Valiant, “A Bridging Model for Parallel Computation,” Communications of the ACM, vol. 33, pp. 103- 111, 1990.

[Vech99] Vivekanand Vellanki and Ann L. Chervenak, “A Cost-Benefit Scheme for High-Performance Predictive Prefetching”, Supercomputing '99, November 1999.

[Vudb01] Richard Vuduc, James Demmel, and Jeff Bilmes, “Statistical Models for Automatic Performance Tuning”, International Conference on Computational Science, San Francisco, CA, USA, May 2001.

[W3arm] ARM software development toolkit, v2.11, Copyright 1996-7, Advanced RISC Machines.

[W3hp04] Hewlett-Packard website, “Loop Transformations”, http://h18009.www1.hp.com/fortran/docs/vf-html/pg/pguoolpt.htm

[W3nc00] NCSA, “SGI MIPSpro Compiler System”, NCSA Archive, http://archive.ncsa.uiuc.edu/SCD/Hardware/Origin2000/Doc/Compiler.html

[Webibm] IBM Corporation, “Engineering Scientific Subroutine Library (ESSL)”, http://www-1.ibm.com/servers/eserver/pseries/library/sp_books/essl.html.

[Weblpk] LAPACK – Linear Algebra Package, http://www.netlib.org/lapack.

[Whal01] R. Clint Whaley, Antoine Petitet, and Jack Dongarra, “Automated Empirical Optimizations of Software and the ATLAS Project”, Parallel Computing, Volume 27, Numbers 1-2, pp 3-25, 2001.

[Wogr02] J. Worringen, A. Gaer, and F. Reker, “Exploiting transparent remote memory access for non-contiguous and one-sided communication,” in proceedings of Workshop for communication architectures in clusters (CAC 02) at IPDPS '02, Fort Lauderdale, FL, 2002.

[Wuwp04] Jiesheng Wu, Pete Wyckoff, Dhabaleswar Panda, “High Performance Implementation of MPI Derived Datatype Communication over InfiniBand,” In Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004.

[Wwds94] M. Weiser, B. Welch, A. Demers, and S. Shenker, “Scheduling for reduced CPU energy”, In 1st Symp. on Operating Systems Design and Implementation (OSDI), November 1994.

[Yasm95] J. C. Yan, S. R. Sarukkai, and P. Mehra, “Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit,” Software Practice & Experience, Vol. 25, No. 4, pp. 429-61, 1995.

[Yeom04] Michael Yeomans, “Factors threaten technology’s amazing potential”, Pittsburgh Tribune-Review, http://pittsburghlive.com/x/tribune-review/business/s_271579.html

[Yvki00] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, “The design and use of SimplePower: A cycle-accurate energy estimation tool,” in Proc. DAC Conf., 2000, pp. 340--345.

[Zhou05] H. Zhou, “Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window”, Proceedings of the 2005 International Conference on Parallel Architectures and Compilation Techniques (PACT’05), 2005.

[Ziso01] Craig Zilles and Gurindar Sohi, “Execution-based prediction using speculative slices”, In Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA), pp. 2-13, 2001.